

Projektarbeit

von:

Winold Doege
Göran Liebig
Mario Kallauke

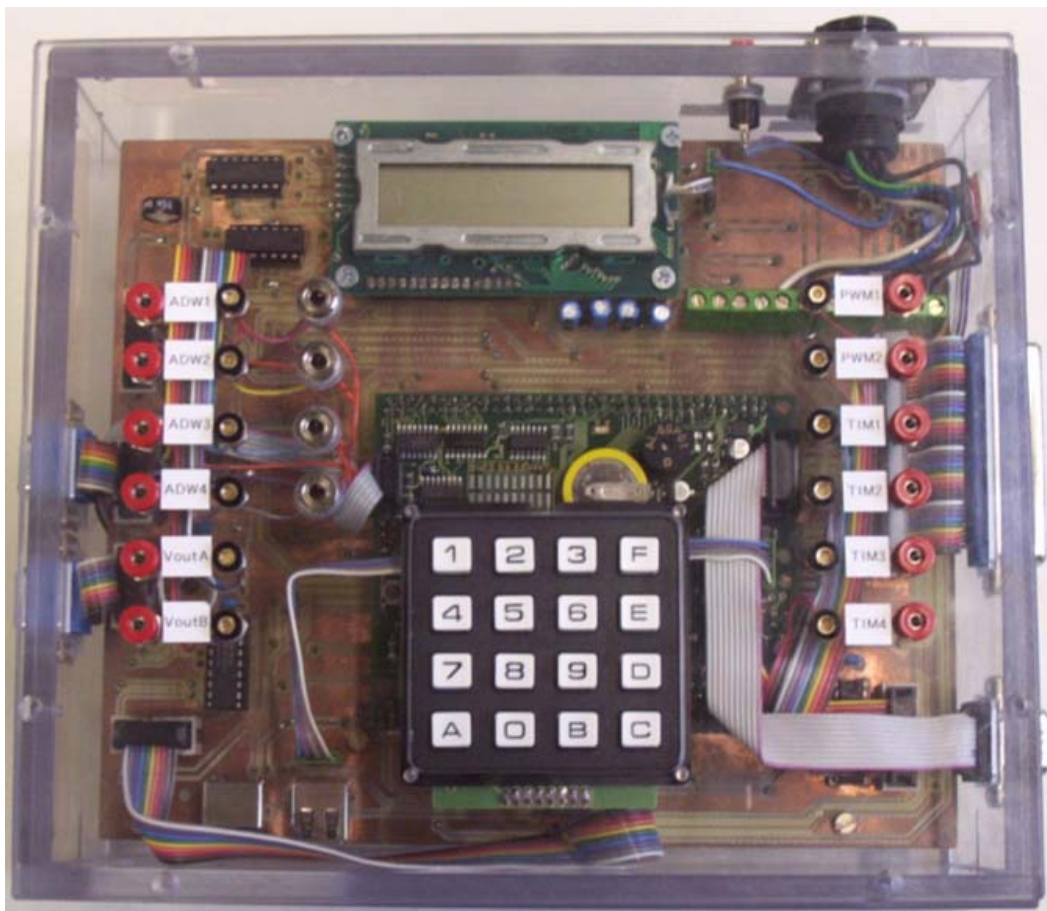
Betreuer:

Herr Stahl

Thema:

**Entwicklung, Inbetriebnahme und Programmierung
eines Mikrocontrollersystems S12Compact
zur Ausbildung an der STB**

Abgabedatum: 27.Mai 2004



Prototyp des Projektes

Inhalt

1.	Vorwort	7
2.	Selbstständigkeitserklärung	8
3.	Aufgabenstellung und Verteilung	9
4.	Quellen / Software	10
5.	Die MCU vom Typ MC9S12DG256	11
5.1	MC9S12DG256	11
5.2	Portbelegung	12
6.	Das S12 Compact Board	15
6.1	Überblick S12compact	15
6.2	Technische Daten	15
6.3	Elektrische Funktion der S12 Compactplatine	16
6.3.1	Stromversorgung	16
6.3.2	Reseterzeugung	17
6.3.3	Takterzeugung und PLL	17
6.3.4	Betriebsarten, BDM-Unterstützung	18
6.3.5	Integriertes EEPROM	18
6.3.6	Indikator-LED	18
6.3.7	Akustischer Signalgeber	19
6.3.8	RS232-Interface	19
6.3.9	USB-Interface	19
6.3.10	D/A-Wandler (DAC)	20
6.3.11	CAN-Interface	20
6.4	Mechanischer Aufbau der Trägerplatine	21
6.5	Elektrischer Aufbau der Trägerplatine	22
6.5.1	Analog / Digital / Wandler	22
6.5.2	Digital / Analog / Wandler	22
6.5.3	CAN Controller	23
6.5.4	LCD Display	23
6.5.5	Piapro	23
6.5.6	Timer / Pulsweitenmodulation	23
6.5.7	Tastatur	23
6.5.8	USB / PS 2	23

7.	Der Einstieg	25
7.1	Blink 1 – Das erste Programm	25
8.	PS/2	26
8.1	Allgemeines	26
8.2	Programmfunktionen	27
9.	RS 232	33
9.1	Allgemeines	33
9.2	Programmfunktionen	34
10.	USB	38
10.1	Allgemeines	38
10.2	Treiber Installation unter Windows XP	38
10.3	Terminal für USB-Schnittstelle	40
10.4	Programmfunktionen	42
11.	Timer	45
11.1	Allgemeines	45
12.	Parport	55
13.	Der Analog Digital Wandler	56
13.1	Allgemeines	56
13.2	Funktionstest/Programmfunktionen/Registerbeschreibungen	57
14.	Digital Analog Wandler	61
14.1	Allgemeines	61
14.2	Funktionstest/Programmfunktionen/Registerbeschreibungen	61
15.	Pulsweitenmodulationsausgänge	64
15.1	Allgemeines	64
15.2	Funktionstest/Programmfunktionen/Registerbeschreibungen	65
16.	Keypad	70
16.1	Allgemeines	70
16.2	Funktionstest/Programmfunktionen/Registerbeschreibungen	70
17.	LCD Display	73
17.1	Allgemeines	73
17.2	Funktionstest/Programmfunktionen/Registerbeschreibungen	75

18.	CAN – Schnittstelle	81
18.1	Überblick	81
18.2	msCAN-Implementation des HCS12-Mikrocontrollers	82
18.3	Sendevorgang des msCAN-Modul	83
18.4	Empfangsvorgang beim msCAN-Modul	84
18.5	Filtern von Botschaften im msCAN	84
18.6	Taktversorgung des msCAN Moduls	85
18.7	Bestimmung der Busting-Parameter	88
18.8	Funktionstest/Programmfunktionen/Registerbeschreibungen	89
19.	Akustischer Signalgeber	102
19.1	Allgemeines	102
19.2	Funktionstest/Programmfunktionen/Registerbeschreibungen	102
20.	LED	105
20.1	Programmfunktionen	105
21.	Interessanter Fehler bei der Projektarbeit	106
22.	Quellcodes	107
	Analog Digital Wandler	
	ATD_test.c (Testprogramm)	107
	s12_atd.c (Funktionen)	109
	Digital Analog Wandler	
	DA_test.c (Testprogramm)	111
	DA.c (Funktionen)	113
	SPI	
	s12_spi.c (Funktionen)	114
	Pulsweitenmodulationsausgänge	
	PWM_test.c (Testprogramm)	114
	s12_PWM_Warnsignale.c (Funktionen)	115
	Keypad	
	Keypad_Lcd_poll.c (Testprogramm)	119
	Keypad_Lcd_int.c (Testprogramm)	120
	Keypad.c (Funktionen)	121

LCD	
Lcd_test.c (Testprogramm)	123
Lcd.c (Funktionen)	124
CAN0	
CAN0_test.c (Testprogramm)	126
CAN0.c (Funktionen)	127
CAN4	
CAN4_test.c (Testprogramm)	129
CAN4.c (Funktionen)	130
Akustischer Signalgeber	
freqout_test.c (Testprogramm)	132
freqout.c (Testprogramm)	133
PIAPORT	
Piaport.c	135
Interruptvektortabelle	
vectors.c	136
Datentypen	
datatypes.h	138
HCS12 I/O	
hcs12dp256.h	139
RS232 / LED	
led_und_rs232_test.c	151
s12_sci.h	152
s12_sci.c	153
USB / LED	
led_und_usb_test.c	154
PS/2	
ps2_test.c	155
s12_ps2.h	156
s12_ps2.c	158

	Timer	
	rechteckgenerator.c	162
23.	Schaltpläne	
	AD Wandler	163
	CAN BUS	164
	LCD	165
	Piapro	166
	Timer	167
	Steckverbinder	168
	Tastatur	169
	USB PS/2	170
	Top Layer	171
	Bottom Layer	172
	Bestückungsplan	173
24.	Bestückungsplan	174

1. Vorwort

Ziel unseres Projektes ist die Entwicklung, der Bau und die Funktionsprüfung einer Platine zur Aufnahme des HCS 12 Compactmodules. Dieser neuartige 16 Bit Mikrocontroller soll im Rahmen der Lehrtätigkeit der Staatlichen Technikerschule eingesetzt werden.

Es ist geplant die vorhandenen HC 11 Module der Staatlichen Technikerschule schrittweise durch die entwickelte Trägerplatine abzulösen, und damit den Unterricht, vor allem der „Automatisierungsklassen“ noch näher an die Bedingungen der Industrie heranzuführen.

Hierzu war es notwendig sich mit den elektrischen Anschlussbedingungen des HCS 12 Moduls, den mechanischen Vorgaben der vorhandenen Unterrichtsmittel, sowie den teilweise völlig neuen Befehlssätzen der Assemblerprogrammierung vertraut zu machen. In enger Zusammenarbeit mit unserem Projektleiter Herrn Stahl und der Elektrowerkstatt der staatlichen Technikerschule wurde bei der Projektentwicklung auch Problematiken wie Ressourcennutzung der vorhandenen elektrischen Bauteile und Möglichkeiten des späteren effektiven Umbaus der Module umfassend angesprochen und gelöst.

Diese Dokumentation kann nur einen kleinen Einblick in die vielfältigen Einsatzmöglichkeiten des HCS 12 geben. Die durchgeführten Programmieraufgaben haben sich stark an den Wissensstand des zu vermittelnden Lehrstoffes orientiert und können deswegen nur ein Wegweiser sein. Es ist sinnvoll die diversen Datenblätter die zum Mikrocontroller gehören zu lesen oder mindestens einen Überblick zu verschaffen. Die Datenblätter sind als PDF-Datei auf der CD-ROM unter Handbücher abgelegt.

Für die Programmierung der vorhandenen Schnittstellen liegt im Anhang jeweils eine funktionstüchtige Quelle vor. Entsprechende Verweise finden sich in den jeweiligen Kapiteln.

Wir hoffen das durch unsere Arbeit viele weitere „Technikergenerationen“ die Möglichkeit erhalten, sich mit der Arbeitsweise des HCS 12 umfassend vertraut zu machen.

2. Selbstständigkeitserklärung

Wir erklären, dass wir die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt haben. Alle Fremdanteile haben wir kenntlich gemacht und auf die Quellen verwiesen.

Winold Doege

Göran Liebig

Mario Kallauke

3. Aufgabenstellung und Verteilung

Winold Doege:	Programmierung der RS 232 -, USB -, PS2 Schnittstelle, Timer, Versuchsaufbauten, Dokumentation, Präsentation
Göran Liebig:	Mech. / Elektrischer Aufbau, Layout, Dokumentation, Präsentation
Mario Kallauke:	Programmierung des ATD, DAC, PWM, Keypad, LCD, CAN0, CAN4, akustischer Signalgeber, Piaport (Lauflicht), Versuchsaufbauten, Dokumentation, Präsentation

4. Quellen

Literatur:	Hinweis
Mikrocontrollerdesign	Standardwerk über den 68HCx12
benutzerhandbuch.pdf	S12 Compact Benutzerhandbuch
s12cov110_schema.pdf	Schaltplan vom S12Compact
CPU12RG.pdf	Reference Guide
S12CPUV2.pdf	Reference Manual
MC9S12DG256PB.pdf	Übersicht vom S12 & Datenblätter
S12ATD10B8CV2.pdf	Analog/Digital Wandler
S12MSCANV2.pdf	CAN Schnittstelle
S12PWM8B8CV1.pdf	PWM
S12SCIV2.pdf	SCI
S12SPIV2.pdf	SPI
S12ECT16B8V1.pdf	Timer
74c922.pdf	Tastatortreiber für Keypad
DRM008.pdf	Alarm Control Panel Designer Reference
dev12_tutorial.pdf	DEV12 Tutorial
timer.pdf	Timerbeschreibung auf Deutsch

Internet:

http://www.elektronikladen.de	
http://www.ftdichip.com	(Treiber für USB Schnittstelle FT232BM)
http://www.motorola.com	(Datenblätter über den S12)
http://govschl.ndsu.nodak.edu/~achapwes/PICmicro/index.html	(PS/2)
http://www.elektronikladen.de/occonsole.html	(OC-Console)
http://www.lemps.ch/tipps/sin50/sinusgenerator.html	(Timer und PWM)

Software:	Version	Hersteller
Orcad	9.2.446 CIS	Cadence Design Systems
Microsoft Word	2002	Microsoft AG
Adobe Acrobat Prof.	6.0.0 19.05.2003	Adobe
Idea12	2.10f & 2.11g	Cosmic
Zap SIM 6812	3.71a	Cosmic
Zap BDM 6812	3.72c	Cosmic
PCAN-View	2.42.4.0	Peak-System Technik
OC-Console		Terminalprogramm
D2xxAPP	1.0	FTDI Ltd.
MC9S12 Unsecure Prog.	1.04	P&E Microcomputer Systems

5. Die MCU vom Typ MC9S12DG256

5.1 MC9S12DG256

Auf dem S12compact kommt eine leistungsstarke MCU vom Typ MC9S12DG256 zum Einsatz. Dieser Mikrocontroller enthält die 16-Bit HCS12 CPU, 256KB Flash, 12KB RAM, 4KB EEPROM und eine große Menge integrierter Peripheriefunktionen, wie SCI, SPI, CAN, IIC, Timer, PWM, ADC und Input-/Output-Kanäle. Der MC9S12DG256 ist vollständig mit 16 Bit breiten internen Datenpfaden ausgestattet. Die integrierte PLL-Schaltung ermöglicht es, Performance und Strombedarf auf einfache Weise den jeweiligen Anforderungen anzupassen.

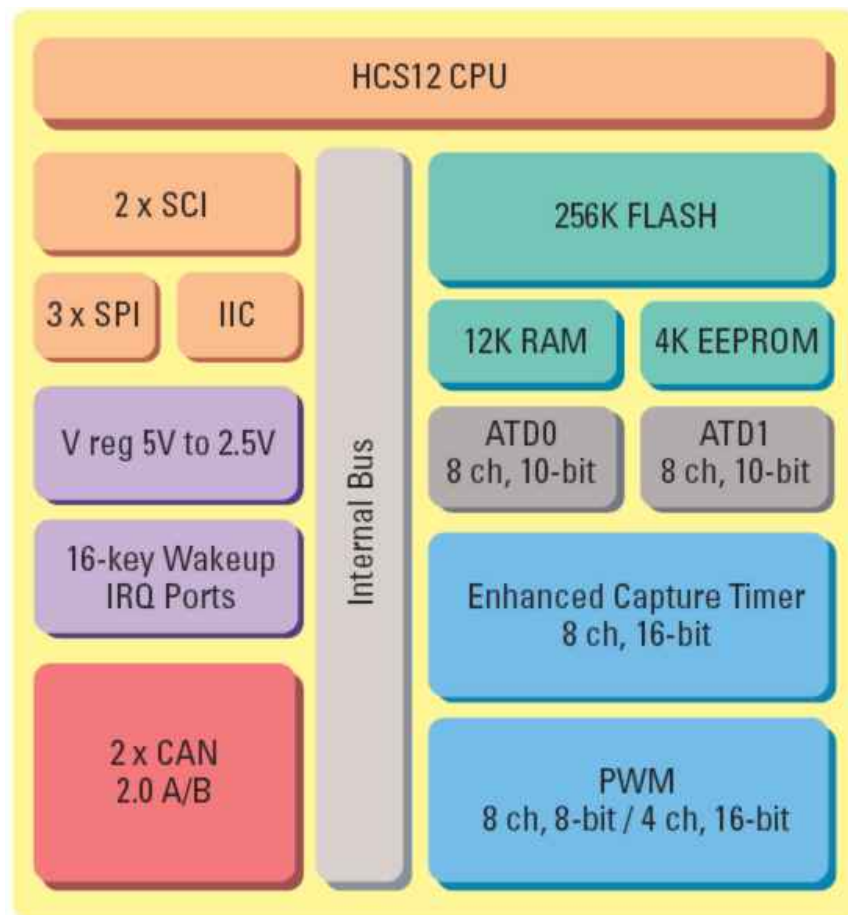


Bild 1: MC9S12DG256

5.2 Portbelegung

Die CPU hat diverse Input und Output Anschlüsse, siehe Blockschaltbild.

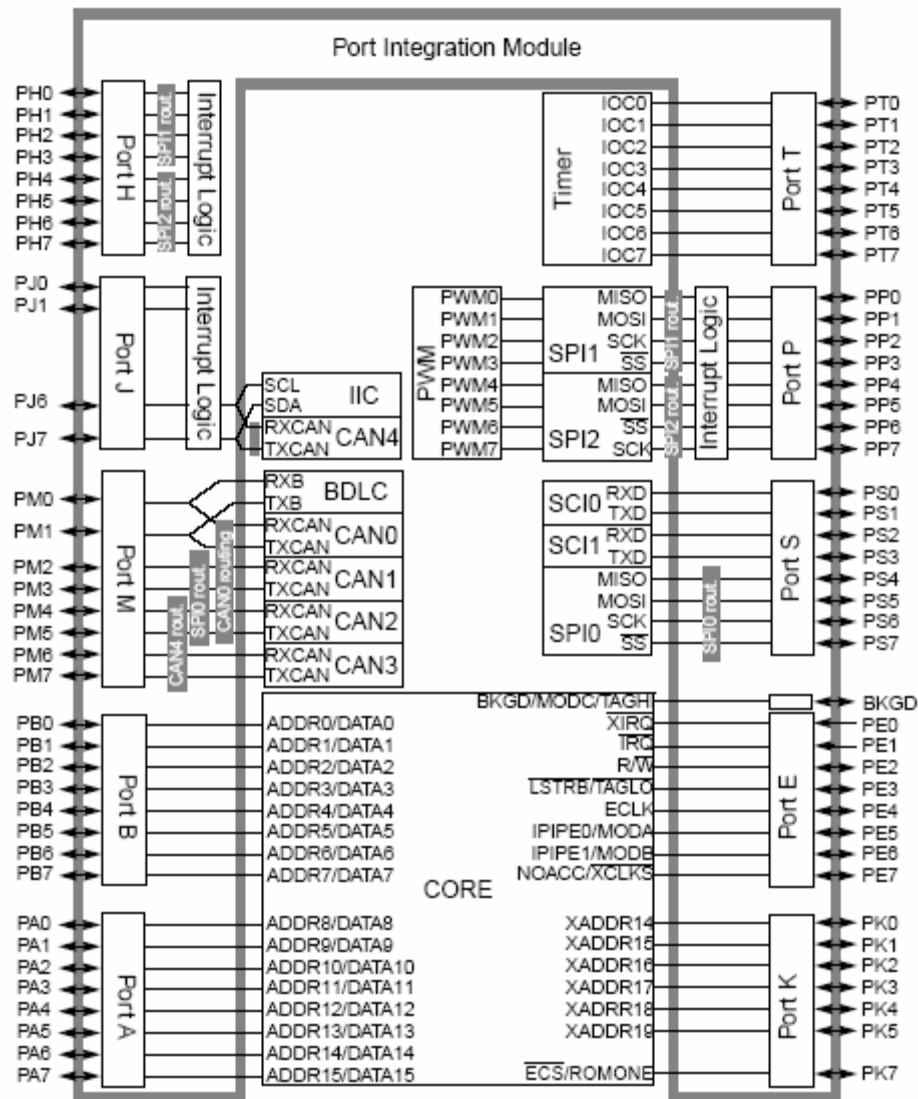


Bild 2: Blockschaltbild S12DP256PIMV2

S12 Port	Port- bezeichnung	Stecker	Bezeichnung	Interrupt fähig	Anschluss für	Anschluss- bezeichnung	Abgesichert über
J	PJ0	X4	PJ0	ja	PS 2 Clock	PJ0	
J	PJ1	X4	PJ1	ja	PS 2 Data	PJ1	
J	PJ6	X4	SDA	ja	CAN RX	SDA	
J	PJ7	X4	SCI	ja	CAN TX	SCI	
H	PH0	X5	MIS01	ja	Piaport	CA1	
H	PH1	X5	MOSI1	ja	Piaport	CA2	
H	PH2	X5	SCK1	ja	Piaport	CB1	
H	PH3	X5	/SS1	ja	Piaport	CB2	
H	PH4	X5	PH4	ja	intern		
H	PH5	X5	PH5	ja	intern		
H	PH6	X5	PH6	ja	intern		
H	PH7	X5	/SPIWR	ja	intern		
K	PK0	X4	PK0	nein	Display	D 0	
K	PK1	X4	PK1	nein	Display	D 1	
K	PK2	X4	PK2	nein	Display	D 2	
K	PK3	X4	PK3	nein	Display	D 3	
K	PK4	X4	PK4	nein	Display	R / W	
K	PK5	X4	PK5	nein	Display	RS	
K	PK7	X5	PK7	nein	Display	E	
B	PB0	X4	PB0	nein	Piaport	PB0	
B	PB1	X4	PB1	nein	Piaport	PB1	
B	PB2	X4	PB2	nein	Piaport	PB2	
B	PB3	X4	PB3	nein	Piaport	PB3	
B	PB4	X4	PB4	nein	Piaport	PB4	
B	PB5	X4	PB5	nein	Piaport	PB5	
B	PB6	X4	PB6	nein	Piaport	PB6	
B	PB7	X4	PB7	nein	Piaport	PB7	
A	PA0	X5	PA0	nein	Piaport	PA0	
A	PA1	X5	PA1	nein	Piaport	PA1	
A	PA2	X5	PA2	nein	Piaport	PA2	
A	PA3	X5	PA3	nein	Piaport	PA3	
A	PA4	X5	PA4	nein	Piaport	PA4	
A	PA5	X5	PA5	nein	Piaport	PA5	
A	PA6	X5	PA6	nein	Piaport	PA6	
A	PA7	X5	PA7	nein	Piaport	PA7	
M	PM0	X4	PM0	nein	intern		
M	PM1	X4	PM1	nein	intern		
M	PM2	X4	PM2	nein	intern		
M	PM3	X4	PM3	nein	intern		
M	PM4	X4	PM4	nein	intern		
M	PM5	X4	PM5	nein	intern		
M	PM6	X5	PM6	nein	intern		
M	PM7	X5	PM7	nein	intern		

S12 Port	Port-bezeichnung	Stecker	Bezeichnung	Interrupt fähig	Anschluss für	Anschluss-bezeichnung	Abgesichert über
P	PP0	X4	PP0	nein	Pulsweiten	Y 1	74LS245
P	PP1	X4	PP1	nein	Pulsweiten	Y 2	74LS245
P	PP2	X4	PP2	nein	Tastatur	D0	
P	PP3	X4	PP3	nein	Tastatur	D1	
P	PP4	X4	PP4	nein	Tastatur	D2	
P	PP5	X4	PP5	nein	Tastatur	D3	
P	PP6	X4	PP6	nein	Tastatur	DA	
P	PP7	X4	PP7	nein	frei		
T	PT0	X4	PT0	nein	Timer out	Y 3	74LS245
T	PT1	X4	PT1	nein	Timer out	Y 4	74LS245
T	PT2	X4	PT2	nein	Timer out	Y 5	74LS245
T	PT3	X4	PT3	nein	Timer out	Y 6	74LS245
T	PT4	X4	PT4	nein	frei		
T	PT5	X4	PT5	nein	frei		
T	PT6	X4	PT6	nein	frei		
T	PT7	X4	PT7	nein	frei		
AD	PAD00	X5	PAD00	nein	ADW	PAD00	LM348
AD	PAD01	X5	PAD01	nein	ADW	PAD01	LM348
AD	PAD02	X5	PAD02	nein	ADW	PAD02	LM348
AD	PAD03	X5	PAD03	nein	ADW	PAD03	LM348
AD	PAD04	X5	PAD04	nein	frei		
AD	PAD05	X5	PAD05	nein	frei		
AD	PAD06	X5	PAD06	nein	frei		
AD	PAD07	X5	PAD07	nein	frei		
AD	PAD08	X5	PAD08	nein	frei		
AD	PAD09	X5	PAD09	nein	frei		
AD	PAD10	X5	PAD10	nein	frei		
AD	PAD11	X5	PAD11	nein	frei		
AD	PAD12	X5	PAD12	nein	frei		
AD	PAD13	X5	PAD13	nein	frei		
AD	PAD14	X5	PAD14	nein	frei		
AD	PAD15	X5	PAD15	nein	frei		

6. Das S12 Compact Board

6.1 Überblick S12compact

S12compact ist eine leistungsfähige Controllerbaugruppe im kompakten Halbeuro-Format auf Basis der 16-Bit Mikrocontrollerfamilie HCS12 von Motorola. Das S12compact Modul erleichtert die Implementierung von umfangreichen Controlleranwendungen, wie z.B. Datenloggerapplikationen. Auf dem S12compact kommt eine leistungsstarke MCU vom Typ MC9S12DG256 zum Einsatz. Dieser Mikrocontroller enthält die 16-Bit HCS12 CPU, 256KB Flash, 12KB RAM, 4KB EEPROM und eine große Menge integrierter Peripheriefunktionen, wie SCI, SPI, CAN, IIC, Timer, PWM, ADC und Input-/Output-Kanäle. Der MC9S12DG256 ist vollständig mit 16 Bit breiten internen Datenpfaden ausgestattet. Die integrierte PLL-Schaltung ermöglicht es, Performance und Strombedarf auf einfache Weise den jeweiligen Anforderungen anzupassen.

Zusätzlich zu den integrierten Controllerfunktionen bietet das S12compact Modul einige interessante Peripherieoptionen. Dazu zählen ein 16-Bit D/A-Wandler nebst Präzisions-Spannungsreferenz, eine batteriegepufferte Echtzeituhr (RTC) und ein USB-Interface. Die für die HCS12-Controller erhältliche umfassende Softwareunterstützung (Monitor, C-Compiler, BDM-Debugger) erleichtert die Entwicklung von Embedded Systemen jeglicher Art.

6.2 Technische Daten

- MCU MC9S12DG256 im LQFP112 Package (SMD)
- HCS12 16-Bit CPU, Programmiermodell und Befehlssatz wie beim HC12
- 16 MHz Quarztakt, bis zu 25 MHz Bustakt über PLL
- 256KB Flash
- 4KB EEPROM
- 12KB RAM
- 2x SCI - asynch. serial Interface (z.B. RS232, LIN)
- 3x SPI - synch. serial Interface
- 1x IIC - Inter-IC-Bus
- 2x msCAN-Module (CAN 2.0A/B-kompatibel)
- 8x 16-Bit Timer (Input Capture/Output Compare)
- 8x PWM (Pulse Width Modulator)
- 16-Kanal 10-Bit A/D-Wandler
- BDM - Background Debug Mode Schnittstelle
- Spezieller LVI-Schaltkreis (Reset Controller)
- BDM-Anschluss für Download und Debugging
- Serielles Interface mit RS232-Treiber, z.B. für PC-Verbindung
- Zweiter serieller Port für IF-Module (RS232, RS485, LIN...)
- Indikator-LED
- Akustischer Signalgeber
- High-Speed phys. CAN-Interface
- Resettaster
- bis zu 70 freie Ein-/Ausgabelleitungen
- acht zusätzliche digitale Eingänge
- acht zusätzliche digitale Ausgänge
- Betriebsspannung 5V, Typ. Stromaufnahme ca. 70 mA
- Abmessungen 80mm x 100mm

Vorhandene Optionale Komponenten

- RTC: batteriegepufferte Echtzeituhr
- DAC: 2-Kanal 16 Bit D/A-Wandler (4096mV)
- USB: Full-speed USB2.0 Interface (belegt zweiten SCI-Kanal)

6.3 Elektrische Funktion der S12 Compactplatine

Hier möchten wir Auszüge des Bedienerhandbuches für das HCS 12 einfügen. Dieser Erklärung ist wegen Ihrer Komplexität nichts zuzufügen. Weitere Daten finden Sie dem beiliegenden Bedienerhandbuch HCS 12, da wir hier nur auf die tatsächlich im Rahmen des Projektes benötigten Baugruppen des HCS 12 eingehen.

6.3.1 Stromversorgung

Der Mikrocontroller (IC1) verfügt über drei Anschlusspaare zur Zuführung der Versorgungsspannung: VDDR/VSSR, VDDX/VSSX und VDDA/VSSA. Die Betriebsspannung beträgt nominal 5 Volt, intern arbeitet der Prozessor jedoch mit 2,5 Volt. Der hierzu erforderliche Spannungsregler ist bereits in der MCU integriert. VREGEN gibt den internen Spannungsregler frei, der Pin ist normalerweise stets mit H-Pegel (5V) zu verbinden. Die Spannungsreduzierung im Core ist in erster Linie erforderlich durch die geringen Strukturbreiten des Fertigungsprozesses (0,25µm und kleiner). Von außen verhält sich der HCS12 jedoch wie ein 5V-Baustein, da an den Ein-/Ausgabepins Pegelwandler vorhanden sind. Eine Ausnahme stellen die Anschlüsse für Oszillator und PLL dar, näheres dazu unten. Die drei genannten Versorgungsanschlusspaare müssen sorgfältig entkoppelt werden. In unmittelbarer Nähe der Pins befindet sich daher je ein 100nF-Keramikkondensator (C15, C16, C17), dem zusätzlich ein 10µF Elektrolytkondensator parallel geschaltet wird (C5, C6, C7). Besonderes Augenmerk muss auf die Entkopplung des VDDA-Pfades gelegt werden, da der interne Spannungsregler aus dieser Spannung seinen Referenzwert (VDDA/2) ableitet. Die interne 2,5-Volt-Corespannung wird an mehreren Stellen nach außen geführt, um sie dort ebenfalls entkoppeln zu können. Hierzu sind an den Anschlusspaaren VDD1/VSS1, VDD2/VSS2 sowie VDDPLL/VSSPLL weitere Keramikkapazitäten vorgesehen (C19, C20, C21). Eine statische Belastung der internen Betriebsspannung durch externe Schaltungskomponenten ist nicht statthaft! Das gilt grundsätzlich auch für VDDPLL, die als Referenzpunkt für die extern angeschlossene PLL-Filterkombination (R3, C3, C4) dient. In die Domäne der Versorgungsspannungen fällt auch die Referenzspannung für die integrierten Analog-Digital-Wandler. Die untere Referenzspannungsgrenze wird über den Anschluss VRL festgelegt, welcher hier (wie meist üblich) auf Massepotential liegt. Die obere Referenzspannung VRH ist über die Lötbrücke BR1 mit VDDA verbunden, C18 dient hier zur Entkopplung. Um die Auflösung der internen 10-Bit A/D-Wandler voll auszuschöpfen, kann eine externe Referenzspannung eingespeist werden. In diesem Fall ist BR1 zu öffnen. VRH darf jedoch VDDA niemals übersteigen. Der TEST-Pin wird nur werkseitig bei Motorola verwendet, in Anwenderschaltungen ist dieser Pin stets mit dem Massepotential zu verbinden.

6.3.2 Reseterzeugung

/RESET ist der bidirektionale, L-aktive Resetpin der MCU. Als Eingang dient er zur Initialisierung der MCU beim Einschalten. Als Open-Drain-Ausgang signalisiert er, dass innerhalb der MCU ein Resetereignis stattgefunden hat. Die HCS12 MCU enthält bereits Schaltungen für Power-On Reset, COP (Watchdog) und Clock Monitor Reset. Es ist dennoch notwendig, zusätzlich einen externen LVI-Schaltkreis vorzusehen, welcher die Aufgabe hat, zuverlässig Reset auszulösen, sobald die Versorgungsspannung der MCU unter den zulässigen Mindestwert gefallen ist.

Der LVI-Schaltkreis IC2 hat einen Open-Drain Ausgang, um Kollisionen mit dem bidirektionalen Resetpin der MCU zu vermeiden. Im inaktiven Zustand stellt sich an /RESET (dank des Pull-Up Widerstands R6) H-Pegel ein. Die Schaltschwelle von IC2 liegt bei typischerweise 4,6V. Das ist geringfügig höher als die Mindestbetriebsspannung der MCU (4,5V). IC2 ist in der Lage, den Resetimpuls auf eine gewisse Mindestlänge auszudehnen, die über den Kondensator C14 festgelegt wird. Bei 100nF beträgt der Delay ca. 50..80ms. Es ist wichtig zu bemerken, dass diese Impulsverlängerung nur bei einem Power-On Reset wirksam wird. Die MCU-internen Resetimpulse werden von IC2 hingegen nicht gedehnt, denn sonst wäre die MCU nicht mehr in der Lage, die korrekte Resetquelle zu ermitteln. Die Konsequenz wäre sonst u.U. ein Programmabsturz durch die Verwendung eines falschen Resetvektors. Es ist daher ebenso wichtig, niemals größere Kapazitäten an die Resetleitung des HCS12 anzuschließen, denn der resultierende Effekt wäre derselbe.

6.3.3 Takterzeugung und PLL

Der On-Chip Oszillator des MC9S12DG256 kann den primären Takt (OSCCLK) mit Hilfe eines Quarzes (Q1) erzeugen, der an die Pins EXTAL und XTAL angeschlossen wird. Der zulässige Frequenzbereich ist 0,5 bis 16 MHz. Wie üblich sind zwei Lastkapazitäten (C1, C2) Teil der Oszillatorschaltung. Die Anordnung ist jedoch modifiziert, wenn man die Schaltung mit der Standard-Pierce Konfiguration vergleicht, wie sie beim HC11 und den meisten HC12-Typen verwendet wurde. Der MC9S12DG256B verwendet einen Colpitts-Oszillator mit translated Ground. Der Hauptvorteil dieser Oszillatorschaltung ist eine sehr geringe Leistungsaufnahme, dafür ist die Komponentenwahl um einiges kritischer. Der S12compact verwendet einen Automotive-Quartz von NDK, mit zwei Lastkapazitäten von lediglich 3,9pF. Darüber hinaus wurde beim Design besonders auf die Minimierung von parasitären Kapazitäten geachtet, die sich nachteilig auf die Signale EXTAL und XTAL auswirken könnten.

Mit einem OSCCLK von 16 MHz ergibt sich ein Default-Bustakt (ECLK) von 8 MHz. Zur Erreichung höherer Taktfrequenzen bedient man sich der PLL-Schaltung des HCS12. Der MC9S12DG256 kann intern mit bis zu 25MHz Bustakt arbeiten, wobei die meisten Designs eine Frequenz von 24MHz nutzen, denn dies ermöglicht eine besonders flexible Festlegung der SCI-Baudraten.

An den Controllerpin XFC wird eine Tiefpassfilterkombination angeschlossen, sie besteht aus den Bauelementen R3, C3 und C4. Ihre Aufgabe ist die Verminderung der Welligkeit des VCO-Signals. Falls die PLL unbenutzt bleibt, kann XFC mit VDDPLL verbunden werden, andernfalls bildet VDDPLL den Bezugspotenzial für den Filter. Die Wahl der Filterkomponenten ist stets ein Kompromiss zwischen Einschwingzeit und Stabilität der Schleife. 5 bis 10kHz Bandbreite und ein Dämpfungsfaktor von 0,9 sind gute Startwerte für die Berechnung. Mit einer Quarzfrequenz von 16MHz und einem gewünschten Busclock von 24MHz ergibt sich eine mögliche Auswahl zu $R3=4,7k$ und $C3=22nF$. $C4$ sollte etwa $(1/20-1/10) \times C3$ betragen, hier also $2,2nF$.

6.3.4 Betriebsarten, BDM-Unterstützung

Drei Pins des HCS12 dienen der Auswahl der MCU-Betriebsart MODA, MODB und BKGD (=MODC). MODA und MODB werden durch die Widerstände R1 und R2 auf L-Pegel gebracht, um Single Chip Mode auszuwählen. BKGD ist über R7 mit H-Pegel verbunden, damit die MCU im Normal Single Chip Mode startet. Dies ist die übliche Betriebsart zur Abarbeitung von Anwendungsprogrammen. Die HCS12 Betriebsart, welche für Download und Debugging genutzt wird, heißt Background Debug Mode (BDM). BDM ist direkt nach Reset aktiv, wenn die MCU im Special Single Chip Mode betrieben wird. Dies wird erreicht, indem - zusätzlich zu MODA und MODB - auch die BKGD-Leitung während Reset vorübergehend auf L-Pegel gebracht wird. Zwischen beiden Modi kann man leicht umschalten, da sich lediglich der Resetzustand der BKGD-Leitung unterscheidet. Ein BDM-Pod, welches am Steckverbinder X1A angeschlossen wird, kann die Umschaltung automatisch vornehmen, und macht einen mechanischen Umschalter überflüssig. Das BDM-Pod wäre ohnehin notwendig zum BDM-basierten Download von Software bzw. als Debugger, gesteuert von Software auf einem (Entwicklungs-) PC. Der 6-pol. Steckverbinder X1A folgt der Motorola-Standardbelegung für BDM12-Anschlüsse. Steckverbinder X1B trägt einige zusätzliche Signale, welche normalerweise für das BDM-Debugging nicht zwingend erforderlich sind. Einige Debugger jedoch bieten zusätzliche Features, welche das Vorhandensein dieser Signale voraussetzen.

6.3.5 Integriertes EEPROM

Der interne EEPROM-Speicher des MC9S12DG256 ist 4KB groß und in 1024 Sektoren zu je 4 Byte (32 Bit) unterteilt. Gelöscht wird stets sektorweise (4 Byte), während die Programmierung wortweise (2 Byte) erfolgen kann. Lesezugriffe auf den EEPROM erfolgen beliebig, also byte- oder wortweise. Nach Reset ist der EEPROM Bereich im MC9S12DG256 ab Adresse 0 gemappt, wird dadurch aber partiell (0x0000.0x03FF) von den Steuerregistern überlagert. Will man nicht auf die ersten 1024 Byte des EEPROM verzichten, muss der EEPROM-Bereich verschoben werden (INITEE Register). Das folgende Beispiel belässt den EEPROM auf der Defaultposition, in der Initialisierungsroutine wird lediglich der EEPROM Clock Divider entsprechend der Quarzfrequenz des S12compact eingestellt. Die Schreibfunktion wrSectEETS() kopiert zwei Worte (4 Byte) von einer beliebigen Quelladresse src auf eine EEPROM-Adresse dest, letztere muss identisch mit einer EEPROM-Sektorgrenze sein (aligned 32 bit). Ist der Inhalt des Zielsektors nicht gelöscht (0xFFFFFFFF), wird zunächst automatisch ein Sector-Erase ausgeführt. Die Zugriffsfunktionen readItemEETS() und writeltemEETS() verallgemeinern den EEPROM-Zugriff dahin gehend, dass nicht mehr mit EEPROM-Adressen gearbeitet wird, sondern mit einer abstrakten Nummerierung von EEPROM-"Items". Jedes dieser EEPROM-"Items" kann 1 bis 4 Byte lang sein.

6.3.6 Indikator-LED

Am Portpin PE6 dient der Pegel des Steuersignals /XCLKS zur Auswahl der Clock-Konfiguration des MC9S12DG256. Führt das Signal H-Pegel, wird der integrierte Colpitts-Oszillator aktiviert. Relevant ist dabei der Zustand zum Zeitpunkt der steigenden Resetflanke. Danach steht PE6 als General-Purpose-I/O zur Verfügung. Auf dem S12compact dient dieses Signal dann zur Ansteuerung der Indikator- LED D2, getrieben vom Buffer IC6C.

6.3.7 Akustischer Signalgeber

Der Signalgeber SP1 wird durch den Buffer IC6D getrieben und wird von der MCU über den Portpin PT2 angesteuert, es sei denn, die Lötbrücke BR3 ist offen. PT2 ist mit einem der acht Timerkanäle der MCU verbunden. Dies ermöglicht die Frequenzerzeugung mit Hilfe der Output-Compare Funktion des Timersystems.

6.3.8 RS232-Interface

Der MC9S12DG256 verfügt über zwei asynchrone Schnittstellen (SCI0, SCI1). Jede dieser Schnittstellen umfasst zwei Signalleitungen (RXDx, TXDx). Handshakeleitungen sind nicht Bestandteil der SCI-Module des Controllers, sie sind durch Einbeziehung zusätzlicher I/O-Ports zu realisieren. SCI0 dient auf dem S12compact als RS232-Interface. Für diese Schnittstelle können auf dem Modul die Ports PM2 und PM3 als Handshakesignale eingesetzt werden. Diese Portsignale lassen sich über die Lötbrücken BR7 und BR5 mit dem RS232-Pegelwandler IC3 verbinden. Ebenso sind die Signalleitungen RXD0 und TXD0 über Brücken (BR6, BR4) mit IC3 verbunden. Öffnet man diese Lötbrücken, können die Controllersignale anderweitig verwendet werden. Sie werden dazu am Steckverbinder X5 bereitgestellt. Der RS232-Anschluß erfolgt über X2. Dieser Steckverbinder ist so gestaltet, dass durch ein Flachbandkabel mit angecrimpter Sub-D9 Buchse eine direkte Verbindung zu einem PC-COM-Port hergestellt werden kann.

6.3.9 USB-Interface

IC15 ist ein USB-UART vom Typ FT232BM. Er realisiert die bidirektionale, transparente Umsetzung einer asynchronen seriellen Schnittstelle in das USB-Protokoll, gemäß der derzeit gültigen USB-Spezifikation 2.0. Der FT232BM gehört der Klasse der USB "Full-Speed" Devices an. Die Daten werden vom HCS12 via SCI1 an den USB-UART übermittelt, dort auf USB umgesetzt und schließlich im PC über einen virtuellen COM-Port bereit gestellt. Die hierzu erforderliche Treibersoftware stellt der Hersteller FTDI für die Anwender des FT232BM abgabefrei zur Verfügung (siehe <http://www.ftdichip.com>). Zum Manuskriptzeitpunkt waren Treiber für Windows-PCs (98 bis XP), Apple-Rechner und Linuxsysteme verfügbar. Auch auf der S12compact Produkt-CD sind die Treiber (Windows-Plattform) enthalten, ggf. existieren aber bereits neuere Versionen. Zwecks Datentransfer zwischen USB-UART und Mikrocontroller sind ausschließlich die Signale TXD und RXD (TXD1 und RXD1 an der MCU) erforderlich. Bei Bedarf kann zusätzlich ein Hardwareprotokoll über /RTS und /CTS implementiert werden. Den /RTS-Ausgang wertet die MCU über ihren Portpin PM6 aus und steuert mit PM7 den /CTS-Eingang des USB-UARTs an. Send- bzw. Empfangsaktivitäten signalisiert der USB-UART bei Bedarf über zwei Leuchtdioden (LEDs). Die Anoden dieser LEDs werden mit VCC (5V) verbunden, die Kathoden an /RXLED bzw. /TXLED angeschlossen (X4/67+68). Über den Anschluss /PWREN zeigt IC15 an, ob die USB-Enumerierung abgeschlossen ist. Dies kann über das Controllersignal PM4 ausgewertet werden. Das Kontrollsignal /SLEEP zeigt an, dass sich IC15 im Suspend-Mode befindet. Über Controllerpin PM5 kann der Zustand dieses Signals ausgewertet werden. Soll durch ein Systemreset auch der USB-UART zurückgesetzt werden, ist BR11 zu schließen. Ein Systemreset bedeutet in diesem Fall stets eine zeitweise Abkopplung des USB-Device vom Bus mit nachfolgender Re-Enumeration. Im Defaultzustand ist BR11 offen, dennoch ist ein korrektes Power-On Reset des USB-UART gewährleistet. Der serielle EEPROM IC16 kann Konfigurationsdaten für den USB-UART enthalten, er ist jedoch im Lieferzustand gelöscht. Der USB-UART antwortet dann auf einen Descriptor-Request mit seinen Standarddescriptoren. User Descriptoren (VID, PID, Strings, Seriennummern etc.) können im EEPROM mit Hilfe eines PC-basierten Utility-

Programms abgelegt werden. Die Programmierung erfolgt in-circuit via USB.
Hinweis: Ist die USB-Option bestückt, steht der IF-Modul Anschluss X3 nicht mehr zur Verfügung.

6.3.10 D/A-Wandler (DAC)

Der 16-Bit D/A-Wandler IC12 ist eine weitere Schaltungsoption des S12compact. Zum Einsatz kommt ein DAC8532 von Burr Brown. Dieser Baustein bietet zwei Kanäle, deren Ausgangsspannungen VOUTA und VOUTB sowohl nacheinander als auch gleichzeitig aktualisiert werden können. IC10 liefert die Referenzspannung (4,096V), welche die obere Ausgangsspannungsgrenze vorgibt. Sie kann durch eine externe Referenzspannungsquelle ersetzt werden kann, wenn zuvor BR9 aufgetrennt wird (Auswirkung auch auf den ADC!). Der Lastwiderstand an den Ausgängen sollte 2kOhm nicht unterschreiten. Beim Einschalten der Betriebsspannung wird die Ausgangsspannung der beiden DAC-Kanäle im IC auf 0V zurückgesetzt. Der DAC wird - wie auch RTC, ADC und die Zusatz-I/Os – über SPI0 angesteuert. /SPICS1 dient als Chipselectsignal für den DAC. Die Ausgangssignale VOUTA und VOUTB sind am Steckverbinder X4 verfügbar.

6.3.11 CAN-Interface

Der MC9S12DG256 verfügt über zwei CAN-Module, die mit CAN0 und CAN4 bezeichnet werden. CAN0 kommuniziert über die Portpins PM0 und PM1 mit einem on-board CAN-Interface Chip (IC5), welcher das physische Businterface bildet. Die CAN-Bussignale CANH und CANL sind dann an X4/63 und X4/64 abzugreifen. Wenn der S12compact der letzte Knoten am CAN-Bus ist, wird eine Terminierung erforderlich. Sie kann durch Schließen der Lötbrücke BR2 aktiviert werden. Das Businterface für CAN4 ist nicht auf dem S12compact realisiert, es muss ggf. extern in der Anwenderschaltung realisiert werden. Die TTL-Signale für CAN4 werden über die Portpins PJ6 und PJ7 geleitet. Hierbei ergibt sich eine Überschneidung mit der IIC-Funktion. Wenn beide Funktionen (IIC und CAN4) genutzt werden sollen, kann man CAN4 mittels MODRR auf PM4/5 oder PM6/7 umleiten, allerdings mit Einschränkungen der Nutzbarkeit des IF-Anschlusses bzw. der USB-Option.

6.4 Mechanischer Aufbau der Trägerplatine

Ziel und Aufgabe des mechanischen Aufbaus der Trägerplatine war die Anpassung der zum Projekt genutzten S12 Compact Platine an das vorhandene Gehäuse für den HC 11. Hier mussten folgende Probleme gelöst werden:

- gleiche mechanische Befestigung im Gehäuse
- Verwendung der vorhandenen Buchsen und Stecker
- Verwendung der vorgehenden Betriebsspannungsversorgung
- Integration von USB und PS 2 Buchse
- Anbindung RS 232 und Tastatur (16 Tasten)
- Integration des zweistelligen Displays
- Integration von einen zusätzlichen CAN Controller
- Schutz der Ein – oder Ausgänge mit entsprechenden Schutzsysteme
- Nutzung der vorhandenen Ein - Ausgangsbuchsen

Weiterhin sollte durch die entsprechende Konstruktion der Trägerplatine der Umbau der vorhandenen Eingabeports zum neuen System erleichtert werden. So ging das Bestreben dahin, die vorhandenen Lötverbindungen zu den D-SUB Verbindern durch Klemmtechnik zu ersetzen. Die vorhandene Stromversorgungsverteilung (extra Leiterplatte) wurde in die Trägerplatine integriert. Auch wurde der Anschluss der Stromversorgungsleitung vom Lötanschluss auf Klemmanschluss geändert. So besteht jetzt die Möglichkeit die Platine ohne die Benutzung eines Lötkolbens komplett zu wechseln.

Durch die Anpassung der Schnittstellenausgänge an das vorhandene Gehäuse wurde erreicht, dass aufwendige Umbauten / Neubauten weitgehend erleichtert oder minimiert wurden.

6.5 Elektrischer Aufbau der Trägerplatine

Der elektrische Aufbau der Trägerplatine diente im Grossen und Ganzen der Anpassung der vorhandenen HCS 12 Platine an die Erfordernisse der Staatlichen Technikerschule. Konkrete Informationen zu den jeweiligen Schnittstellen entnehmen Sie bitte den jeweiligen Kapiteln.

6.5.1 Analog / Digital / Wandler

Projektvorgabe war die Realisierung von 4 Analog / Digital / Wandlereingängen. Diese Eingänge sollten zum HCS 12 Port AD (PAD 00 – PAD 03) geführt werden. Außerdem sollte zur Absicherung der Mikrocontrollereingänge eine Schutzschaltung vorgesehen werden, welcher bei Fehlern durch Studierende auch schnell und preiswert ersetzt werden kann. Die Eingänge sollten von außen zugänglich sein.

Zum Einsatz kam ein Operationsverstärker LM 348 (4x OPV). Dieser Baustein ist neben der vorhandenen Integration von gleich 4 Operationsverstärkern auch aus Kostengründen gewählt wurden. Der Operationsverstärker wurde als Impedanzwandler (Spannungsfolger) aufgebaut. Diese Schaltung weist vor allem einen sehr hohen Eingangswiderstand und eine Verstärkung von „1“ auf.

Der Spannungsfolger nutzt einen „unerwünschten Nachteil“ des Operationsverstärkers, die Anstiegsgeschwindigkeit aus. Durch eine Erhöhung der Eingangsspannung würde selbige an den Ausgang weitergereicht werden und als Rückkopplung auf den invertierenden Eingang diesen gegenläufig steuern. Somit wurden bei idealen Operationsverstärkern extrem hochfrequente Rechteckspannungen entstehen.

Bei realem Operationsverstärker kann die Ausgangsspannung dem Eingang nicht schnell genug folgen und pendelt sich auf einen „Mittelwert“ ein, welcher „fast“ genau der Eingangsspannung folgt.

Als Absicherung der Operationsverstärkereingänge wurden jeweils zwei Schottkydioden BAT 42 eingesetzt. Sie sollen durch ihre Schnelligkeit und Schaltung den Eingang vor Spannungen über VCC und negativen Spannungen schützen. Außerdem besteht die Möglichkeit durch Bestückung des Tiefpasses R 1 und C 1 eine Unterdrückung von Frequenzen oberhalb der Abtastrate des ADC zu erreichen. In der Schaltung ist eine Kombination von 8,2 KOhm und 470 pFarad vorgesehen, was eine Frequenz von 259 kHz bedeuten würde.

6.5.2 Digital / Analog / Wandler

Für den Digital / Analog / Wandler gilt selbe Funktionsweise, nur das auf die Dioden verzichtet werden konnte, da keine fehlerhafte Spannung erwartet wird. Außerdem ist durch den Operationsverstärker der Ausgang des Mikrocontrollers gegen Überlast und Kurzschluss geschützt. Auch können kleine Lasten gegebenenfalls direkt angeschlossen werden. Das RC Glied dient der Glättung der Ausgangskurve.

6.5.3 CAN Controller

Der MC9S12DG256 verfügt über zwei CAN-Module, die mit CAN0 und CAN4 bezeichnet werden. CAN0 kommuniziert über die Portpins PM0 und PM1 mit einem on-board CAN-Interface Chip (IC5), welcher das physische Businterface bildet. Wenn der S12compact der letzte Knoten am CAN-Bus ist, wird eine Terminierung erforderlich. Sie kann durch schließen der Lötbrücke BR2 aktiviert werden. Das Businterface für den CAN4 ist nicht auf dem S12compact realisiert. Dieses haben wir auf unserer Trägerplatine durch den CAN-Interface Chip PCA82C250T realisiert. Die TTL Signale für CAN4 werden über die Portpins PJ6(SDA) und PJ7(SCL) geleitet.

Durch die Steckbrücke J2 kann auch hier eine Terminierung vorgenommen werden. Der CAN Controller wird im Kapitel CAN BUS sehr ausführlich erklärt.

6.5.4 LCD Display

Zum Einsatz kam ein zweizeiliges HD 44780. Dieses sehr häufig eingesetzte Display sollte nach Möglichkeit direkt auf der Platine integriert werden. So können Anschlussfehler und Verluste durch lose Displays weitestgehend eingeschränkt werden.

Versorgt wird über die vorhandenen VCC Spannung. Helligkeitsanpassung erfolgt über das Poti R 9. Bei dem eingesetzten Display ist der LED – Anschluss für die Hintergrundbeleuchtung separat vorgenommen wurden, welches durch einen separaten Anschluss (J6) jederzeit aktiviert werden kann.

6.5.5 Piaport

Hier erfolgt eine reine „Durchschaltung“ der Port A / B des Mikrocontrollers (PA0 bis PA7 & PB0 bis PB7) . Außerdem sollte eine Spannungsversorgung an die entsprechenden Anschlüsse erfolgen. Die Spannungen sind mit Kondensatoren zur Glättung abgesichert.

6.5.6 Timer / Pulsweitenmodulation

Projektvorgabe war der Anschluss von zwei Pulsweitenmodulationsausgängen und von 4 Timerausgängen. Weiterhin sollten die Ausgänge gegen Kurzschluss und Überlast geschützt werden.

Dieser Schutz erfolgt mit dem unidirektionalen Treiberbaustein 74 LS 541. Hier werden die jeweiligen Signale „einfach“ durchgereicht. Die lowaktiven Gateanschlüsse wurden an Masse geführt, womit der Baustein nicht mehr gesondert selektiert werden muss.

Bei Kurzschluss oder Überlast wird nur der Treiber zerstört, nicht aber der Controller.

6.5.7 Tastatur

Hier wurde ein Tastaturcontroller 74C922 eingesetzt. Dieser Baustein kann 4 Zeilen x 4 Spalten auswerten und kann damit 16-stellige Tastaturen ansteuern. Mit den Kondensatoren C 19 und C 20 wird die interne Oszillatorfrequenz des 74C922 eingestellt, mit der er die Tastatur abprüft.

Der 74C922 setzt den „Data av“ (DA) Ausgang auf ,1' wenn ein Tastendruck erfolgt ist.

Achtung: Es wird nur der letzte Tastendruck gespeichert.

Wird nun sein „OE“ (Output-enable) Eingang vom Prozessor auf Null gesetzt, so gibt der IC die Information (4 Bit) auf den Bus. In unserer Anwendung wurde der Chip ständig aktiviert. (lowaktives OE auf Masse)

6.5.8 USB / PS 2

Hier erfolgte nur eine Anpassung der USB Schnittstelle des HCS 12 an eine Standard USB Buchse. Die PS 2 Schnittstelle erhielt ebenfalls zwei Porteingänge und zwei Pulldown Widerstände zur Erkennungsmöglichkeit für den Anschluss der Tastatur.

7. Der Einstieg

7.1 Blink 1 – Das erste Programm

Für das allererste Mikrocontrollerprogramm ist es sinnvoll, einen Ausgabekanal zu verwenden, der auf Anhieb funktioniert. Ohne langes Nachdenken über die richtigen Initialisierungsschritte. Was wäre da geeigneter als eine Leuchtdiode!

Auf dem Mikrocontrollerboard HCS12 Compact ist eine solche Leuchtdiode vorhanden. Sie ist an der Leitung PORTE[7] des Mikrocontrollers angeschlossen, d.h. am obersten Bit des Port E.

In unserem Funktionstest lassen wir diese LED blinken und verwenden dafür das folgende Testprogramm „blink1.c“.

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor(en): Mario Kallauke, Göran Liebig, Winold Doege
// Datei: BLINK1.c
// Datum: 2004-02-12
// Funktion: LED auf s12Compact blinken lassen
//=====

#include "hcs12dp256.h"

//-----

void delay(void)
{
    unsigned long n = 200000;
    while(--n) ;
}

//-----

void main(void)
{
    DDRE |= 0x80;
    while(1)
    {
        PORTE ^= 0x80;
        delay();
    }
}
//=====
```

8. PS/2

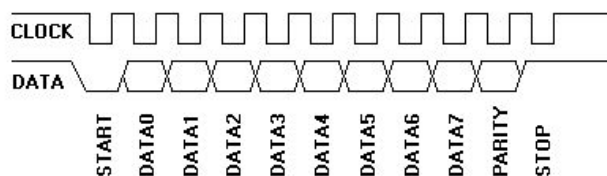
8.1 Allgemeines

Die PS/2 Schnittstelle ist preiswerte Möglichkeit eine Tastatur / Maus an den Controller zu betreiben. Die Tastatur wird dazu über zwei digitale I/O-Portleitungen an das System angeschlossen. Man spart dadurch eine Menge I/O-Anschlüsse. Die Daten der PS/2-Schnittstelle werden serial übertragen. Die Tastatur erzeugt dafür ein Clock Signal. Das bedeutet, der Mikroprozessor muss sich nach dem Taktsignal der Tastatur richten. Tastatur → Master; Mikroprozessor → Slave. Die Daten werden im Format eines UART-Zeichens übertragen.

- Serielles Datenformat
- Startbit (immer 0),
- 8-Bit Daten/ Scancode (LSB zuerst),
- Paritätsbit (ungerade Quersumme),
- Stoppbit am Zeichenende (immer 1),
- Signal zwischen Zeichen immer 1:

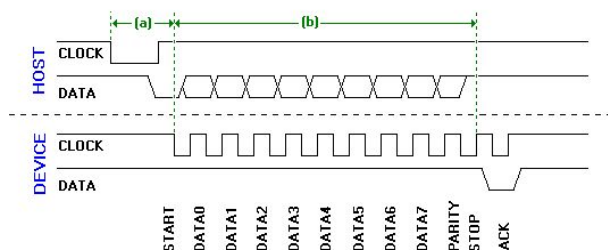
Bei den acht Daten-Bits ist das LSB zuerst und MSB zuletzt. Das Taktsignal kann je nach Tastatur zwischen 10kHz und 16.7 kHz schwanken. Die Zeichen die übertragen werden bezeichnet man als Scan-Codes. Es gibt drei Scan-Codes.

Datentransfer Tastatur → Mikroprozessor



Datentransfer Mikroprozessor → Tastatur

Um die Datenübertragung vom Host zur Tastatur etwas besser zu erklären ist der Transfer in in zwei Teile aufgeteilt:



- a) Der Host ist der Mikrocontroller und Device ist die Tastatur. Der Ausgang vom Host zieht die Clockleitung für mindestens 100ms auf low. Am Ende der 100ms zieht man die Datenleitung auf low. Die Clockleitung wird dann wieder losgelassen und wird so wieder high. Das ist das Startsignal.

- b) Die Tastatur erkennt diesen Sendewunsch vom Host. Die Tastatur setzt die Clockleitung auf low. Jetzt kann der Host das erste Bit (0 oder 1) auf die Datenleitung legen. Das Bit wird mit der negativenflanke übernommen. Nach der Übernahme wird die Clockleitung wieder high. Das nächste Bit wird auf die Datenleitung gelegt. Das gleiche passiert mit den anderen 7 Bits und dem Parity Bit. Das Stop - Bit ist in diesem Fall high kurz darauf wird von der Tastatur eine Bestätigung (ACK-Signal low) auf die Datenleitung gelegt. Dieses Signal sollte vom Host ausgewertet werden. Wenn keine Bestätigung erfolgt ist ein Fehler passiert und es sollte noch einmal übertragen werden.

8.2 Programmfunktionen

void init_ps2(void)

Aufgabe:	Initialisierung der PS/2 Schnittstelle
Header:	s12_ps2.h
Funktionsaufruf:	init_ps2()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Diretion Register von Port J	0xFD	PJ1 auf Eingang
PIEJ	PortJ nur lesen	0x02	Datasignal löst dann Interrupt aus

void set_CLKDAT_ON(void)

Aufgabe:	Setzt Clock- und Datasignal auf high
Header:	s12_ps2.h
Funktionsaufruf:	set_CLKDAT_ON()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Diretion Register von Port J	0x03	PJ0 und PJ1 auf Ausgang
PTJ	PortJ Register	0x03	Setzt Clock- und Datasignal auf high

void set_CLKDAT_OFF(void)

Aufgabe:	Setzt Clock- und Datasignal auf low
Header:	s12_ps2.h
Funktionsaufruf:	set_CLKDAT_OFF()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Diretion Register von Port J	0x03	PJ0 und PJ1 auf Ausgang
PTJ	PortJ Register	0xFC	Setzt Clock- und Datasignal auf low

void clock_on(void)

Aufgabe:	Setzt Clocksignal auf high
Header:	s12_ps2.h
Funktionsaufruf:	clock_on()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Diretion Register von Port J	0x01	PJ0 auf Ausgang
PTJ	PortJ Register	0x01	Setzt Clocksignal auf high

void clock_off(void)

Aufgabe:	Setzt Clocksignal auf low
Header:	s12_ps2.h
Funktionsaufruf:	clock_off()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Direction Register von Port J	0x01	PJ0 auf Ausgang
PTJ	PortJ Register	0xFE	Setzt Clocksignal auf low

void data_on(void)

Aufgabe:	Setzt Datasignal auf high
Header:	s12_ps2.h
Funktionsaufruf:	data_on()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Direction Register von Port J	0x02	PJ1 auf Ausgang
PTJ	PortJ Register	0x02	Setzt Datasignal auf high

void data_off(void)

Aufgabe:	Setzt Datasignal auf low
Header:	s12_ps2.h
Funktionsaufruf:	data_off()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Direction Register von Port J	0x02	PJ1 auf Ausgang
PTJ	PortJ Register	0xFD	Setzt Datasignal auf low

void set_clock(UINT8 Bit)

Aufgabe:	Setzt je nach Wert Clocksignal 0 oder 1
Header:	s12_ps2.h
Funktionsaufruf:	set_clock(UINT8)
Übergabeparameter:	Bitwert 0 oder 1 setzen
Rückgabeparameter:	keine
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Diretion Register von Port J	0x01	PJ1 auf Ausgang
PTJ	PortJ Register	0 oder 1	Setzt Datasignal den Bitwert

void set_data(UINT8 Bit)

Aufgabe:	Setzt je nach Wert Datasignal 0 oder 1
Header:	s12_ps2.h
Funktionsaufruf:	set_data(UINT8)
Übergabeparameter:	Bitwert 0 oder 1 setzen
Rückgabeparameter:	keine
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Diretion Register von Port J	0x02	PJ1 auf Ausgang
PTJ	PortJ Register	0 oder 1	Setzt Datasignal den Bitwert

int read_clock(void)

Aufgabe:	Lese Clocksignal
Header:	s12_ps2.h
Funktionsaufruf:	read_clock()
Übergabeparameter:	keine
Rückgabeparameter:	Bitwert 0 oder 1
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Diretion Register von Port J	0x01	PJ0 auf Eingang
PTIJ	PortJ nur lesen Register	0 oder 1	liest Clocksignal den Bitwert

int read_data(void)

Aufgabe:	Lese Datasignal
Header:	s12_ps2.h
Funktionsaufruf:	read_data()
Übergabeparameter:	keine
Rückgabeparameter:	Bitwert 0 oder 1
Reccourcen:	keine

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRJ	Data Diretion Register von Port J	0x02	PJ1 auf Eingang
PTIJ	PortJ nur lesen Register	0 oder 1	Datasignal den Bitwert

unsigned char kb_out(void)

Aufgabe:	Byte von der Tastatur einlesen
Header:	s12_ps2.h
Funktionsaufruf:	kb_out()
Übergabeparameter:	keine
Rückgabeparameter:	unsigned char Byte
Reccourcen:	keine

void kb_in(UINT8 byte)

Aufgabe:	Byte in die Tastatur schreiben
Header:	s12_ps2.h
Funktionsaufruf:	kb_in(UINT8)
Übergabeparameter:	Bytewert
Rückgabeparameter:	keine
Reccourcen:	keine

interrupt void kb_isr(void)

Aufgabe:	Byte auf dem Bildschirm über Interrupt ausgeben
Header:	s12_ps2.h
Funktionsaufruf:	in Vectortabelle „kb_isr“ unter der Adresse 0xFFCE Port J eintragen
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	Port J Pin 0 und 1

void _initPLL (void)

Aufgabe:	PLL auf die gewünschte CPU Taktfrequenz einstellen
Header:	s12_crg.h
Funktionsaufruf:	iniPLL()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	keine

Auszug aus der Header Datei S12_crg.h

```
.
.
.
#define S12_OSCCLK      16000000          // input frequency from Xtal/Osc
#define S12_REFCLK      8000000          // PLL internal reference clock
#define S12_PLLCLK      48000000         // 24MHZ
// #define S12_PLLCLK    32000000         // 16MHZ
// #define S12_PLLCLK    16000000         // 8MHZ

// PLL clock generation formula, according to CRG Block User Guide:
// PLLCLK = OSCCLK * 2(SYNR+1) / (REFDV+1)
//
#define S12_REFDV ((S12_OSCCLK/S12_REFCLK)-1)
#define S12_SYNR ((S12_PLLCLK/(2*S12_REFCLK))-1)

#define S12_ECLK (S12_PLLCLK/2) // final bus clock frequency (ECLK)
// #define S12_ECLK (S12_OSCCLK/2) // bus clock if PLL not in use / off
```

9. RS 232

9.1 Allgemeines

Die RS232 ist die Standard Schnittstelle zwischen PC und der Außenwelt. Hier ist ein kleines Beispiel, wie man die Verbindung zwischen PC und unserem S12Compact Modul herstellt.



HyperTerminal starten

Start → Alle Programme → Zubehör → Kommunikation
→ HyperTerminal (Klick)

Namen z.B. „S12Compact“ eingeben und → OK

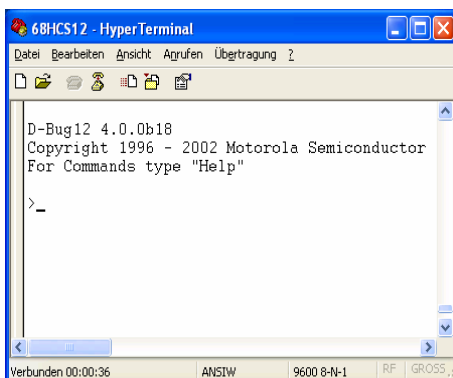


Überprüfen Sie an welchen Port (COM1 oder COM2) am PC der S12Compact angeschlossen ist.

Dann die Schnittstelle einstellen und → OK



Die Schnittstelle wie auf dem Bild einstellen und → OK



Wenn man jetzt den S12compact neu startet bzw. ein „Reset“ auslöst kommt diese Meldung.

9.2 Programmfunktionen

Die RS232 Schnittstelle und die dazugehörigen Funktionen. Eine Übersicht des Aufbaues befindet sich am Ende von 9.2.

void initSCI0(UINT16 bauddiv)

Aufgabe:	Initialisierung der SCI Schnittstelle
Header:	s12_sci.h
Funktionsaufruf:	initSCI0(UINT16 bauddiv)
Übergabeparameter:	Baudratenteiler = Taktfrequenz : 16 : Baudrate
Rückgabeparameter:	Keine
Reccourcen:	RS-232 (Port S : RXD0 , TXD0)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
SCIBD	Teiler	Bauddiv & 0x1fff	Initialisierung der Baudrate
SCICR1	Modus	0x00	0 → 8N1
SCICR2	Ein- und Ausgang normal, kein Interrupt	0x0C	Sender + Empfänger einstellen

Registeraufbau: BASIS Adresse ist 0xC8

Register address: \$_0

	7	6	5	4	3	2	1	0
R	0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8
W								
RESET:	0	0	0	0	0	0	0	0

Register address: \$_1

	7	6	5	4	3	2	1	0
R								
W	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
RESET:	0	0	0	0	0	1	0	0

 = Unimplemented or Reserved

Figure 3-2 SCI Baud Rate Registers (SCI BDH/L)

13 Bit für die Baudraten Einstellung.

Register address: \$_2

	7	6	5	4	3	2	1	0
R								
W	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT
RESET:	0	0	0	0	0	0	0	0

 = Unimplemented or Reserved

Figure 3-3 SCI Control Register 1 (SCICR1)

Um den Modus einzustellen.

Register address: \$_3

	7	6	5	4	3	2	1	0
R	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
W								
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-4 SCI Control Register 2 (SCICR2)

TE — Transmitter Enable Bit

TE enables the SCI transmitter and configures the TXD pin as being controlled by the SCI. The TE bit can be used to queue an idle preamble.

1 = Transmitter enabled

0 = Transmitter disabled

RE — Receiver Enable Bit

RE enables the SCI receiver.

1 = Receiver enabled

0 = Receiver disabled

UINT8 getSCI0(void)

Aufgabe:	Ein Byte von der RS232-Schnittstelle lesen.
Header:	s12_sci.h
Funktionsaufruf:	UINT8 getSCI0()
Übergabeparameter:	Keine
Rückgabeparameter:	Ein Byte von der RS-232-Schnittstelle.
Reccourcen:	RS-232 (Port S : RXD0 , TXD0)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
SCISR1	Ist etwas da?	0x20	Holt Wert aus SCIDRL, wenn 0
SCI0DRL	Register wo das Byte liegt	0x??	Wird ausgelesen

Register address: \$_4

	7	6	5	4	3	2	1	0
R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
W								
RESET:	1	1	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-5 SCI Status Register 1 (SCISR1)

Register address: \$_6

	7	6	5	4	3	2	1	0
R	R8		0	0	0	0	0	0
W		T8						
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Register address: \$_7

	7	6	5	4	3	2	1	0
R	R7	R6	R5	R4	R3	R2	R1	R0
W	T7	T6	T5	T4	T3	T2	T1	T0
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-7 SCI Data Registers (SCIDRH/L)

void putSCI0(UINT8)

Aufgabe:	Ein Byte zur RS232-Schnittstelle schicken
Header:	s12_sci.h
Funktionsaufruf:	putSCI0(UINT8)
Übergabeparameter:	Ein Byte was auf der RS-232 Schnittstelle gesendet wird.
Rückgabeparameter:	Keine
Reccourcen:	RS-232 (Port S : RXD0 , TXD0)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
SCISR1	Ist das vorherige Byte schon weg?	0x80	Schreibt Wert in das SCIDRL , wenn 0
SCIDRL	Register wo das Byte hingeschrieben wird	0x??	Wird geschrieben

Register address: \$_4

	7	6	5	4	3	2	1	0
R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
W								
RESET:	1	1	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-5 SCI Status Register 1 (SCISR1)

Register address: \$_6

	7	6	5	4	3	2	1	0
R	R8		0	0	0	0	0	0
W		T8						
RESET:	0	0	0	0	0	0	0	0

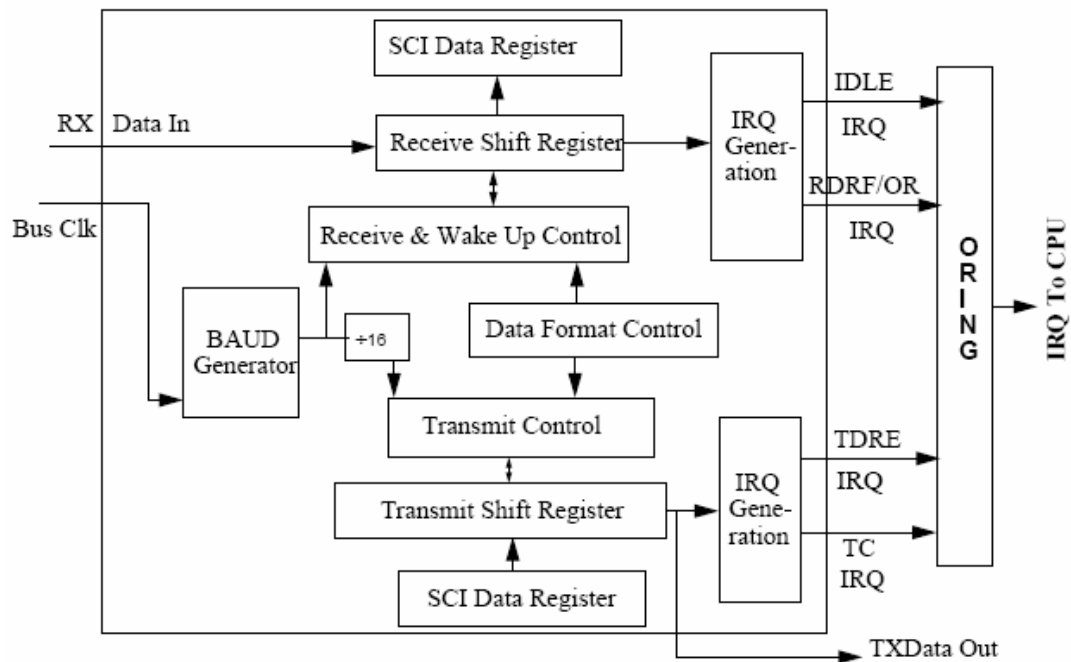
= Unimplemented or Reserved

Register address: \$_7

	7	6	5	4	3	2	1	0
R	R7	R6	R5	R4	R3	R2	R1	R0
W	T7	T6	T5	T4	T3	T2	T1	T0
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-7 SCI Data Registers (SCIDRH/L)



Blockschaltbild der SCI Schnittstelle

Offset	Use	Access
\$_0	SCI Baud Rate Register High (SCIBDH)	Read/Write
\$_1	SCI Baud Rate Register Low (SCIBDL)	Read/Write
\$_2	SCI Control Register1 (SCICR1)	Read/Write
\$_3	SCI Control Register 2 (SCICR2)	Read/Write
\$_4	SCI Status Register 1 (SCISR1)}	Read
\$_5	SCI Status Register 2(SCISR2)	Read/Write
\$_6	SCI Data Register High (SCIDRH)	Read/Write
\$_7	SCI Data Register Low (SCIDRL)	Read/Write

BASIS Adresse ist 0xC8 für SCI 0 und 0xD0 für SCI 1

Register Quick Reference

Register name	Bit 7	6	5	4	3	2	1	Bit 0	Addr. offset
SCIBDH	Read: 0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8	\$_0
SCIBDL	Read: SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0	\$_1
SCICR1	Read: LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT	\$_2
SCICR2	Read: TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	\$_3
SCISR1	Read: TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	\$_4
SCISR2	Read: 0	0	0	0	0	BRK13	TXDIR	RAF	\$_5
SCIDRH	Read: R8	T8	0	0	0	0	0	0	\$_6
SCIDRL	Read: R7	R6	R5	R4	R3	R2	R1	R0	\$_7
	Write: T7	T6	T5	T4	T3	T2	T1	T0	

= Reserved or unimplemented

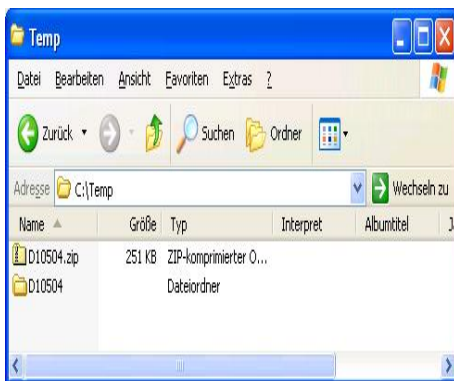
10. USB

10.1 Allgemeines

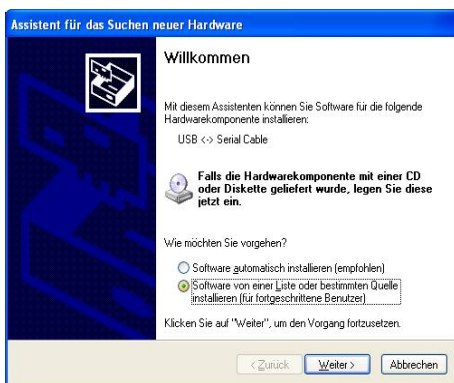


Auf dem S12Compact Modul ist dieser Chip für die USB Kommunikation zuständig.

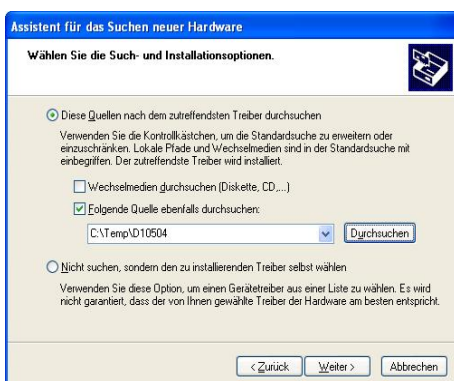
10.2 Treiber Installation unter Windows XP



Die Datei „D10504.zip“ in das Verzeichnis c:\temp kopieren und entpacken.



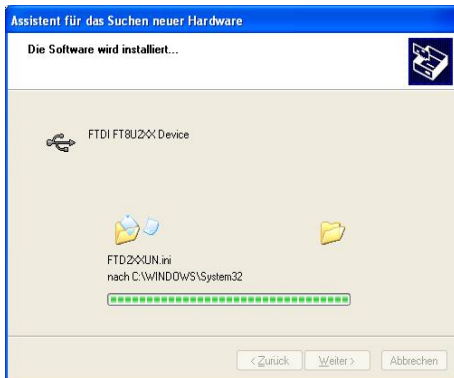
Den USB Stecker vom S12Compact Modul mit dem PC verbinden.



Pfad direkt eintippen oder über „Durchsuchen“ anwählen.



Laufwerksbuchstabe und Verzeichnis auswählen und OK drücken.



Treiber wird in das Verzeichnis C:\Windows\ System32 kopiert.

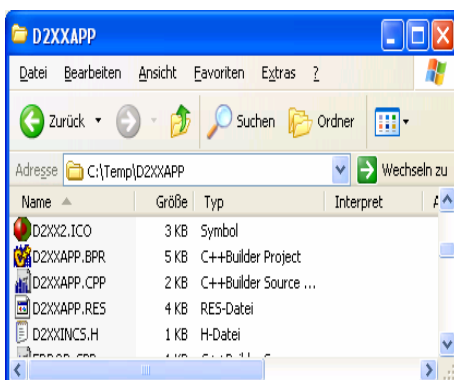


Fertig stellen drücken.

10.3 Terminal für USB-Schnittstelle



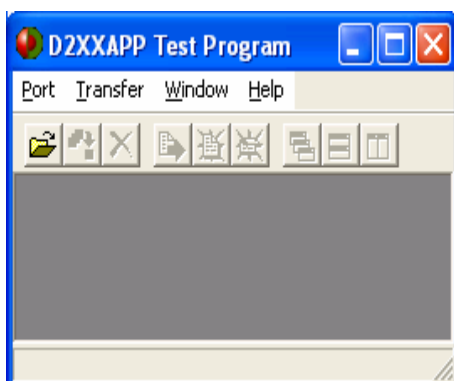
Die Datei „D2XXAPP.ZIP“ in das Verzeichnis c:\Temp kopieren und entpacken.



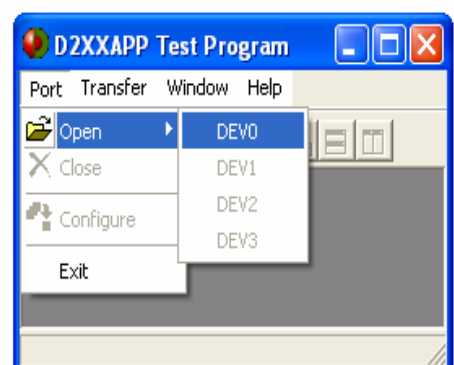
Im Verzeichnis befindet sich die Datei „D2SSAPP.BPR“. Ist der Borland Builder noch nicht installiert bitte jetzt installieren.

Wenn der Borland Builder installiert ist → Datei öffnen. Projekt kompilieren und starten (F9).

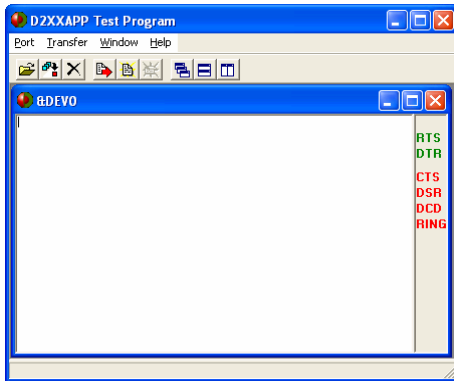
Die Datei „d2xxapp.exe“ wird erzeugt.



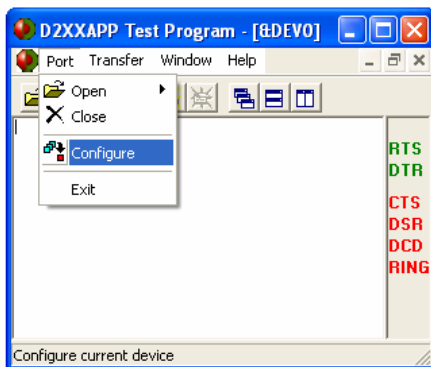
So sieht das fertige Programm aus.



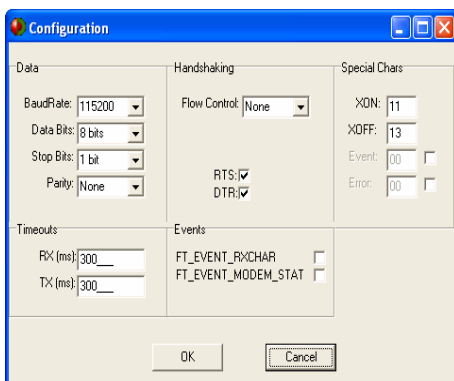
über Port → Open → DEV0 kann man das bereits angeschlossene S12Compact Modul auswählen.



Es öffnet sich ein weiteres Fenster.



Über Port → Configure kann man die Geschwindigkeit einstellen.



10.4 Programmfunktionen

Die USB Schnittstelle und die dazugehörigen Funktionen.

void initSCI1(UINT16 bauddiv)

Aufgabe:	Initialisierung der SCI Schnittstelle
Header:	s12_sci.h
Funktionsaufruf:	initSCI1(UINT16 bauddiv)
Übergabeparameter:	Baudratenteiler
Rückgabeparameter:	Keine
Reccourcen:	USB (Port S : RXD1 , TXD1)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
SCIBD	Teiler	bauddiv & 0x1fff	Initialisierung der Baudrate
SCICR1	Modus	0x00	0 → 8N1
SCICR2	Ein- und Ausgang normal, kein Interrupt	0x0C	Transmitter + Receiver aktiv

Registeraufbau: BASIS Adresse ist 0xC8

Register address: \$_0

	7	6	5	4	3	2	1	0
R	0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8
W								
RESET:	0	0	0	0	0	0	0	0

Register address: \$_1

	7	6	5	4	3	2	1	0
R	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
W								
RESET:	0	0	0	0	0	1	0	0

= Unimplemented or Reserved

Figure 3-2 SCI Baud Rate Registers (SCI BDH/L)

13 Bit für die Baudraten Einstellung.

Register address: \$_2

	7	6	5	4	3	2	1	0
R	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT
W								
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-3 SCI Control Register 1 (SCICR1)

Um den Modus einzustellen.

Register address: \$_3

	7	6	5	4	3	2	1	0
R	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
W								
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-4 SCI Control Register 2 (SCICR2)

TE — Transmitter Enable Bit

TE enables the SCI transmitter and configures the TXD pin as being controlled by the SCI. The TE bit can be used to queue an idle preamble.

1 = Transmitter enabled

0 = Transmitter disabled

RE — Receiver Enable Bit

RE enables the SCI receiver.

1 = Receiver enabled

0 = Receiver disabled

UINT8 getSCI1(void)

Aufgabe:	Ein Byte von der RS232-Schnittstelle lesen.
Header:	s12_sci.h
Funktionsaufruf:	UINT8 getSCI1()
Übergabeparameter:	Keine
Rückgabeparameter:	Ein Byte von der USB-Schnittstelle.
Reccourcen:	USB (Port S : RXD1 , TXD1)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
SCISR1	Ist etwas da?	0x20	Holt Wert aus SCIDRL, wenn 0
SCI0DRL	Register wo das Byte liegt	0x??	Wird ausgelesen

Register address: \$_4

	7	6	5	4	3	2	1	0
R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
W								
RESET:	1	1	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-5 SCI Status Register 1 (SCISR1)

Register address: \$_6

	7	6	5	4	3	2	1	0
R	R8	T8	0	0	0	0	0	0
W								
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Register address: \$_7

	7	6	5	4	3	2	1	0
R	R7	R6	R5	R4	R3	R2	R1	R0
W	T7	T6	T5	T4	T3	T2	T1	T0
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-7 SCI Data Registers (SCIDRH/L)

void putSCI1(UINT8)

Aufgabe:	Ein Byte zur RS232-Schnittstelle schicken
Header:	s12_sci.h
Funktionsaufruf:	putSCI1(UINT8)
Übergabeparameter:	Ein Byte was auf der USB Schnittstelle gesendet wird.
Rückgabeparameter:	Keine
Reccourcen:	USB (Port S : RXD1 , TXD1)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
SCISR1	Ist das vorherige Byte schon weg?	0x80	Schreibt Wert in das SCIDRL , wenn 0
SCIDRL	Register wo das Byte hingeschrieben wird	0x??	Wird geschrieben

Register address: \$_4

	7	6	5	4	3	2	1	0
R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
W								
RESET:	1	1	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-5 SCI Status Register 1 (SCISR1)

Register address: \$_6

	7	6	5	4	3	2	1	0
R	R8	T8	0	0	0	0	0	0
W								
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Register address: \$_7

	7	6	5	4	3	2	1	0
R	R7	R6	R5	R4	R3	R2	R1	R0
W	T7	T6	T5	T4	T3	T2	T1	T0
RESET:	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-7 SCI Data Registers (SCIDRH/L)

11. Timer

11.1 Allgemeines

Das Timerport (PORTT) des HC(S)12 kann für verschiedene Timerfunktionen genutzt werden. z.B. als Rechteck- und/oder Sinusgenerator mit RC-Glied. Die Timer kann man auch als PWM benutzen. Über die Software kann für jeden einzelnen PIN festgelegt werden, ob er als Input Capture oder als Output Compare genutzt werden soll. Für den Pulsweitenmodulator stehen die Kanäle 0-6 zur Verfügung. Werden die Timerfunktionen nicht benötigt, so lässt sich das PORTT als normales digitales I/O Port nutzen. Der Timer setzt den Status des Datenrichtungsregisters DDRT außer Kraft. Der Timer ist ein 16Bit Zähler (0 - 65535).

Der Timerausgang PT2 ist gleichzeitig am Buzzer angeschlossen. Durch öffnen der Lötbrücke BR3 ist die „Hupe“ deaktiviert.

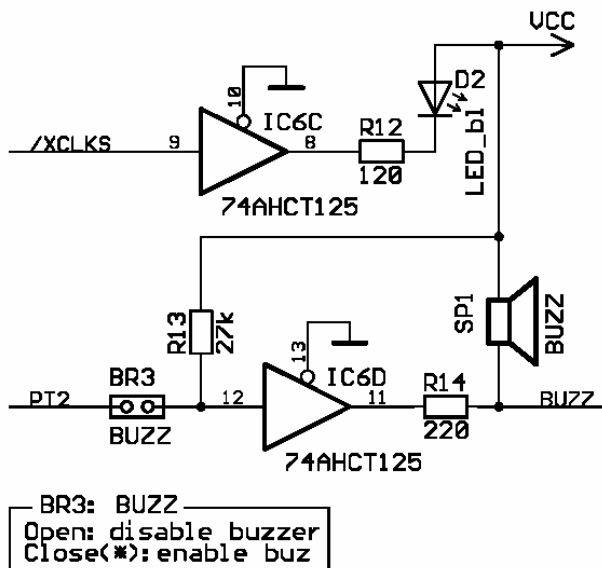


Bild 1: PT2 Hupe

Die Bezeichnungen haben sich geändert:	
alt	neu
TSCR	TSCR1
TMSK2	TSCR2
TMSK1	TIE
OC7M	TOC7M
OC7C	TOC7C

Es gibt verschiedene Energiespar-Modi.

STOP: Der Timer ist aus und alle Takte sind gestoppt.
 FREEZE: Der Hauptzähler läuft weiter.
 WAIT: Der Hauptzähler läuft weiter.
 NORMAL: Der Hauptzähler läuft weiter.

Pulsweitenmodulator (PWM)

Mit dem Output Compare lässt sich beim HC12 ein Pulsweitenmodulator realisieren, der ohne den Prozessor zu belasten, betrieben werden kann. Der Output Compare Kanal 7 hat gegenüber den anderen Kanälen eine Sonderstellung: Wird im Timer Interrupt Mask 2 Register das TCRE-Bit gesetzt ist, wird bei einem Output Compare am Kanal 7 das Timer Count Register auf '0' zurückgestellt. Das bedeutet, dass mit dem Kanal 7 die Periodendauer des Rechtecksignales festgelegt wird, die anderen sieben Kanäle lassen sich als PWM verwenden. Im OC7D wird eingestellt, welchen Zustand (High/Low) die einzelnen OC-Kanäle nach einem Output Compare am Kanal 7 haben sollen. Durch die Werte in den Timer Input Capture/Output Compare Registern 0-6 lassen sich die Pulsweiten regulieren. Wichtig ist dass die Timer Control Register 1/2 so eingestellt sind, dass sich der Zustand am Ausgang der PWM-Kanäle sich nach einem Output Compare ändert.

Pulse Accumulator

Pulse Accumulator heißt auf Deutsch etwa soviel wie Impulszähler. Der HC12 Pulse Accumulator kann allerdings einiges mehr, als einfach Impulse zu zählen. Der Pulse Accumulator ist wie der Timer Counter ein 16-Bit Zähler, der, falls im Time Accumulation Mode laufend, über den Timer Prescaler angesteuert wird. Die Einstellung dieses Teilers hat allerdings keinen Einfluss, der Pulse Accumulator wird immer über einen Teiler durch 64 betrieben. Ein Takt des Pulse Accumulator Clocks (PACLK) dauert beim Lems12 also 8_s. Der Pulse Accumulator kann als Vorteiler für den Timer Counter verwendet werden, damit lassen sich sehr lange Zeiten am Timer Counter generieren. Im Time Accumulation Mode zählt der Pulse Accumulator die Zeit, während der der Kanal 7 des Timer Counters auf High oder Low liegt. Im Event Counter Mode zählt der Pulse Accumulator bei jeder am Kanal 7 ankommenden, fallenden oder ansteigenden Flanke um eins vorwärts.

Wichtige Register für den Pulse Accumulator:

- TSCR1 (Timer System Control Register1): Ohne das das Timer Enable Bit gesetzt ist, arbeitet der Pulse Accumulator nur im Event Counter Modus!
- PACTL (Pulse Accumulator Control Register): In diesem Register werden alle wichtigen Einstellungen für den Betrieb des Pulse Accumulators vorgenommen.
- PAFLG (Pulse Accumulator Flag Register): Nach ein Überlauf des Pulse Accumulator Counters oder wenn die im PACTL eingestellte Flanke am Timerkanal 7 eingetroffen ist, werden hier die entsprechenden Interrupt Flags gesetzt.
- PACNT (Pulse Accumulator Count Register): Der Zähler des Pulse Accumulators. Im Gegensatz zum Timer Counter ist hier ein Schreibzugriff jederzeit möglich.

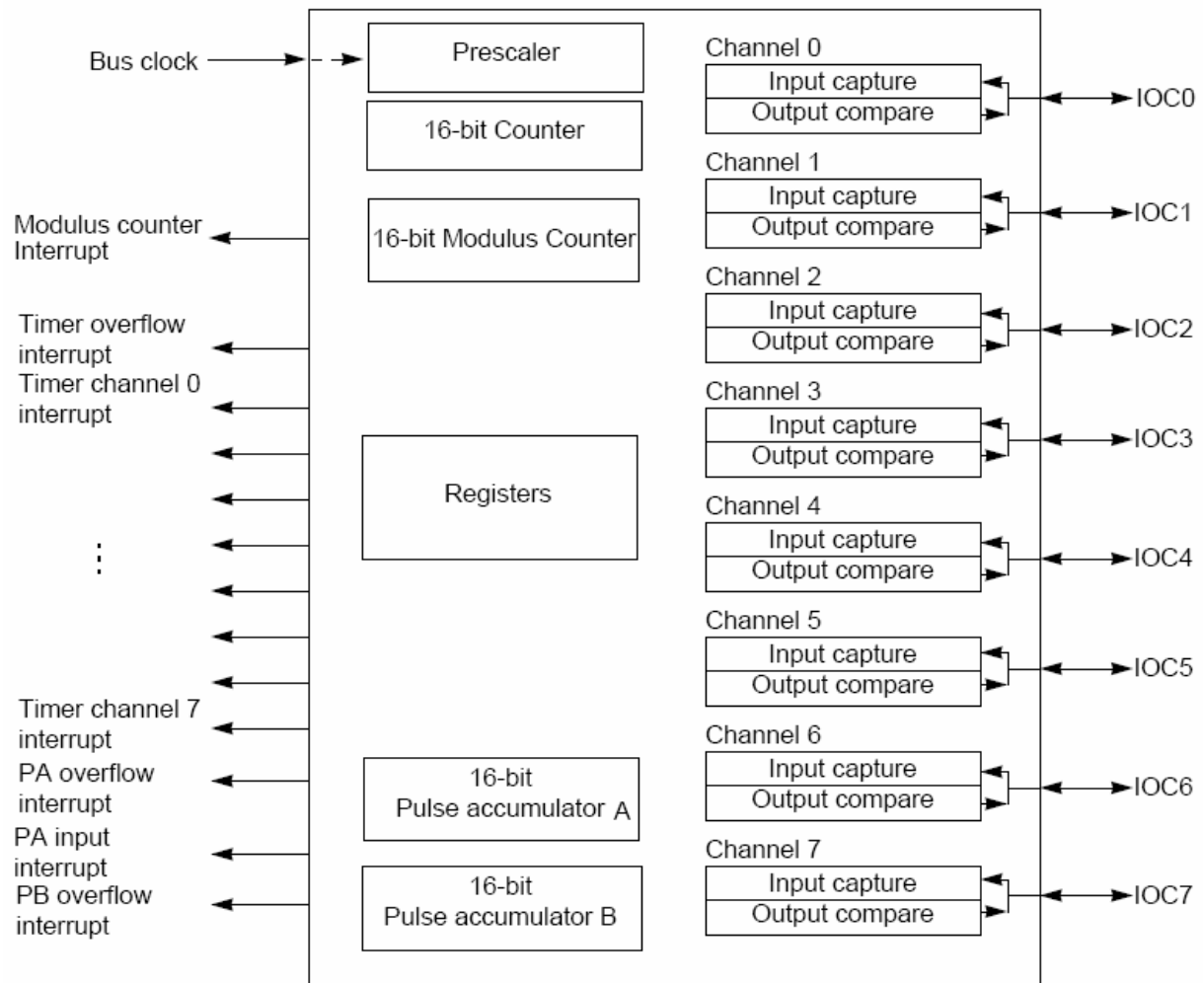


Bild 2: Timer Block Diagram HCS12

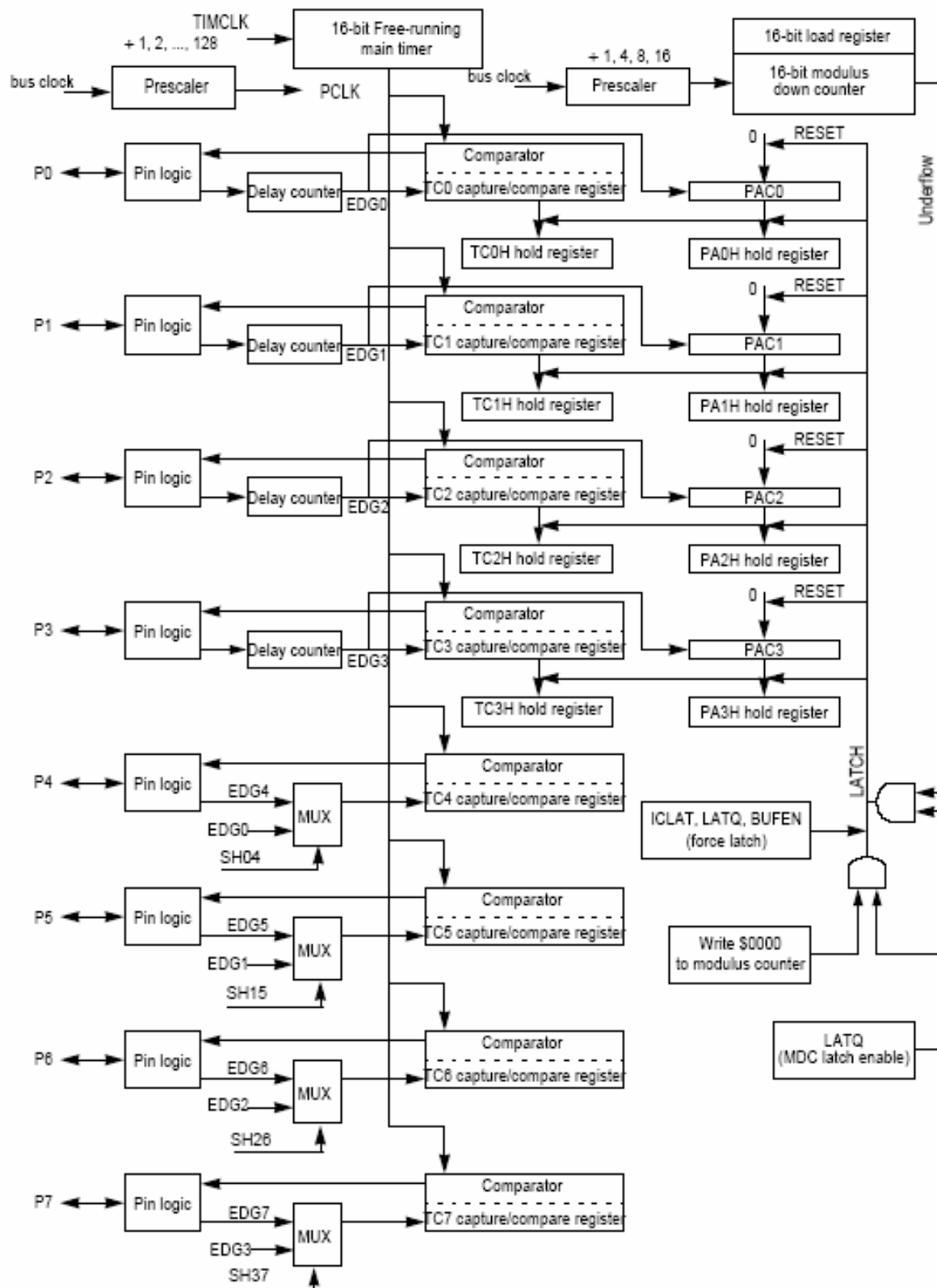


Bild 3: Detailed Timer Block Diagram in Latch mode

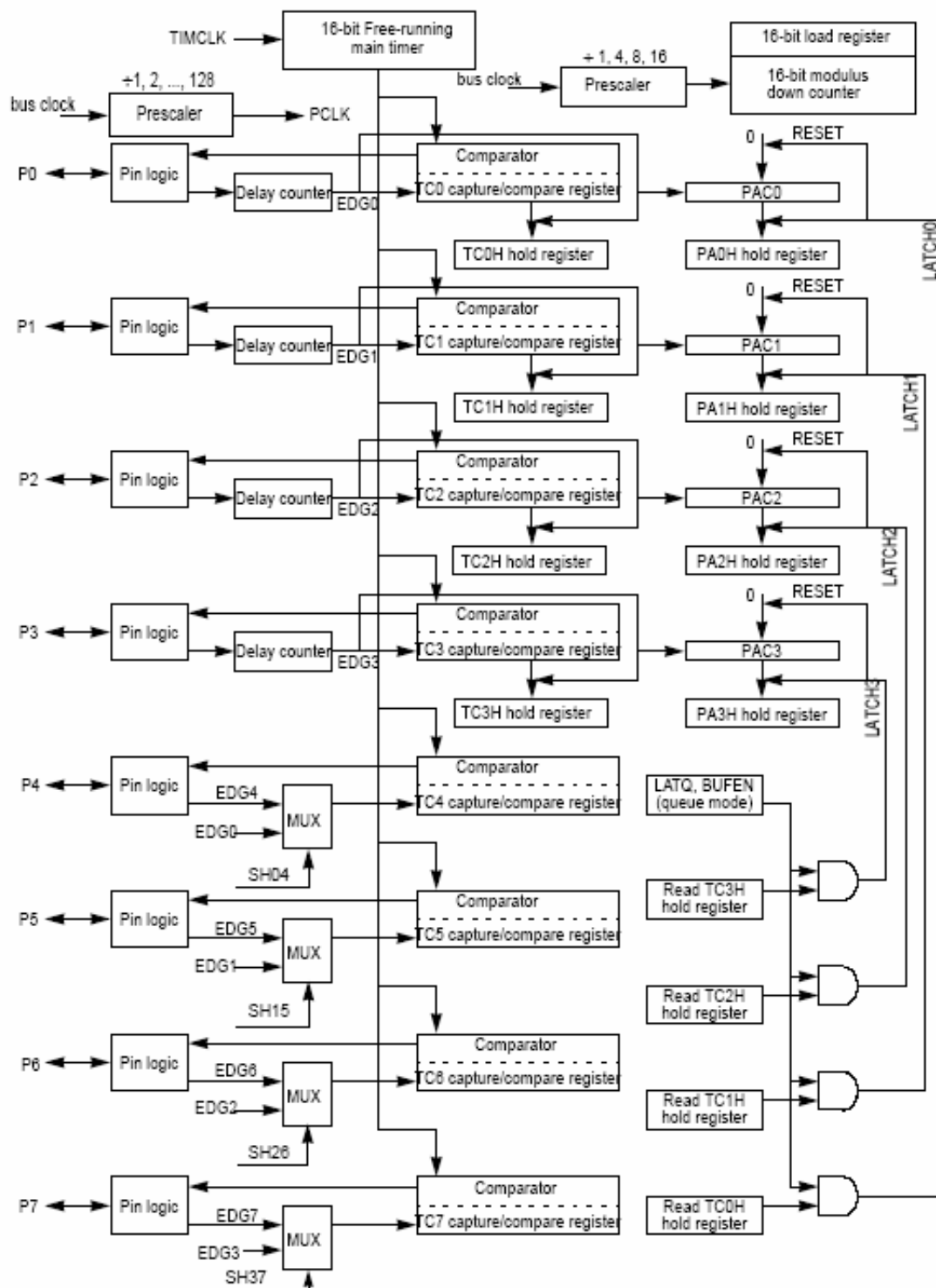


Bild 4: Detailed Timer Block Diagram in Queue mode

Input Capture

Input Capture bedeutet auf Deutsch übersetzt etwa soviel wie Eingangserfassung. Sämtliche Kanäle des Timerports lassen sich als Input Capture nutzen. So stehen im Maximum acht Kanäle zur Verfügung. Über das TIOS werden die einzelnen Kanäle als Input Capture oder als Output Compare selektiert. In den Timer Control Register 3 und 4 wird für jeden PIN des PORTT einzeln festgelegt, auf welche Flanke des Eingangssignals, der Timer reagieren soll. Zu jedem einzelnen Kanal gehört ein Timer Input Capture/Output Compare Register. Sobald die gewünschte Flanke am Eingang ankommt, wird der Wert des Timer Count Registers in das zugehörige Timer Input Capture Register geschrieben. Gleichzeitig wird das Timer Interrupt Flag gesetzt. Durch dieses Flag kann der Programmierer, beziehungsweise das Programm überprüfen, ob die gewünschte Flanke bereits eingetroffen ist, oder nicht. Ist dieses Flag gesetzt, so wird keine weitere Prüfung des Eingangssignals mehr vorgenommen. Dazu muss dieses Flag von der Software wieder zurückgesetzt werden. Im TIE kann für die einzelnen Kanäle festgelegt werden, ob nach einem Input Capture ein Interrupt ausgelöst werden soll, oder nicht. Für Berechnungen wie Perioden- oder Einschaltdauer werden nach dem setzen des Interrupt Flags die Werte aus dem Input Capture Register ausgelesen. Diese Werte können nachher für Berechnungen gebraucht werden.

Timer Control Register 3/4 (TCTL3/4)

Die Timer Control Register 3 und 4 sind die einzigen Register, in denen die Timerfunktionen eingestellt werden, welche für die Input Capture Funktion wichtig sind. In diesen beiden Registern wird für die einzelnen als Input Capture geschalteten Kanäle festgelegt, auf welche Flanke am Kanaleingang ein Input Capture ausgelöst werden soll.

Output Compare

Auf Deutsch übersetzt bedeutet Output Compare etwa soviel wie Ausgangsvergleicher. Wird einer oder mehrere der Timerkanäle 0-7 als Output Compare gesetzt, wird dauernd der Inhalt, des zum Kanal gehörenden Timer Input Capture/ Output Compare Registers, mit dem Timer Counter Register verglichen. Sobald die beiden Inhalt gleich sind wird im TFLG1 das Input Capture/Output Compare 'n' Flag gesetzt. Im TIE kann für die einzelnen Kanäle festgelegt werden, ob nach einem Output Compare ein Interrupt ausgelöst werden soll oder nicht. Damit lassen beispielsweise Timer programmieren. Es empfiehlt sich in den meisten Fällen den Vorteiler des Timer Count Registers im TSCR2 so einzustellen, dass der Zählintervall 1_s beträgt. Damit lassen sich nachher am einfachsten Berechnungen ausführen.

Wichtige Register für den Output Compare:

- CFORC (Timer Compare Force Register): Über dieses Register kann per Software ein Output Compare an einem als Output Compare geschalteten Kanal ausgelöst werden.
- TOC7M (Timer Output Compare 7 Mask Register): Dieses Register hat für die als Output Compare geschalteten Kanäle dieselbe Bedeutung wie die Datenrichtungsregister für die normalen I/O-Ports.
- TOC7D (Timer Output Compare 7 Data Register): Hier wird festgelegt, welchen Zustand die einzelnen Output Compare Kanäle, die in TOC7M als Ausgänge geschaltet wurden, nach einem Output Compare an Kanal 7 haben.
- TCTL1/2 (Timer Control Register 1/2): In diesen beiden Registern wird für die einzelnen als Output Compare geschalteten Kanäle festgelegt, was an dessen Ausgang nach einem Output Compare passieren soll.

Register, mit denen die Timerfunktionen gesteuert werden:

- TIOS (Timer Input Capture/Output Compare Select): Hier wird für die einzelnen Timerkanäle festgelegt, ob sie als Input Capture oder als Output Compare verwendet werden sollen.
- TSCR1 (Timer System Control Register 1): Im TSCR1 wird der Timer aktiviert und festgelegt, ob nach einem Zugriff auf eines der Timer Input Capture/Output Compare Register das Timer Interrupt Flag im TFLG1 gelöscht wird.
- TIE (Timer Interrupt Enable Register): In diesem Register wird für jeden einzelnen Kanal festgelegt, ob nach einem Input Capture oder einem Output Compare (je nach Einstellung im TIOS) ein Interrupt ausgelöst wird.
- TSCR2 (Timer System Control Register 2): Unter anderem wird hier der Timer Prescaler (Zeiteiler) eingestellt, es wird hier festgelegt, ob nach einem Output Compare an Kanal 7 der Counter (Zähler) zurückgestellt werden soll, und ob nach einem Timer Überlauf ein Interrupt ausgelöst werden soll. (Bit7 TOI)
- TFLG1 (Timer Interrupt Flag 1): Das Interrupt Flag wird gesetzt, wenn sich am betreffenden Kanal ein Input Capture oder Output Compare ereignet hat.
- TFLG2 (Timer Interrupt Flag 2): Wird gesetzt, wenn sich ein Überlauf des Zählers (0xFFFF nach 0x0000) ereignet hat.

TIOS (Timer Input Capture/Output Compare Select)

Im Timer Input Capture/Output Compare Register wird festgelegt, welche PIN's des PORTT als Input Capture, beziehungsweise als Output Compare, verwendet werden sollen. Dieses Register kann jederzeit ausgelesen oder neu beschrieben werden. Alle Kanäle, deren Bits '0' gesetzt sind, werden als Input Capture genutzt, alle anderen Kanäle als Output Compare.

Register offset: \$_00

	BIT7	6	5	4	3	2	1	BIT0
R	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0
W	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0
RESET:	0	0	0	0	0	0	0	0

TIOS Register

TSCR1 (Timer System Control Register 1)

Über dieses Register wird der Timer ein und ausgeschaltet. Das Ein und Ausschalten des Timers ist jederzeit möglich. Wenn der Timer ausgeschaltet ist bedeutet dies, dass der 16-Bit Zähler außer Betrieb gesetzt ist. Normalerweise muss nur das Timer Enable Bit gesetzt werden.

Register offset: \$_06

	BIT7	6	5	4	3	2	1	BIT0
R	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0
W								
RESET:	0	0	0	0	0	0	0	0

TSCR1 Register**TIE** (Timer Interrupt Mask)

Im Timer Interrupt Mask wird für jeden einzelnen Timerkanal festgelegt, ob nachdem das Timer Interrupt Flag gesetzt worden ist, ein Interrupt ausgelöst werden soll, oder nicht. Nach einem Reset sind diese Bit alle auf '0' gesetzt, das heißt, das Auslösen eines Interrupts ist deaktiviert. Um die Interruptfunktion zu aktivieren, müssen die Bits der betreffenden Kanäle '1' gesetzt werden. Lese und Schreibzugriffe auf dieses Register sind jederzeit möglich.

Register offset: \$_0C

	BIT7	6	5	4	3	2	1	BIT0
R	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I
W								
RESET:	0	0	0	0	0	0	0	0

TIE Register

CnI: Timer Input Capture/Output Compare "x" Interrupt Enable

TSCR2 (Timer System Control Register 2)

Im Timer Interrupt Mask 2 werden verschiedene Einstellungen, die für den Betrieb des Timers von Bedeutung sind, vorgenommen. Hier kann festgelegt werden, ob bei einem Überlauf des Timers (von 65535 nach 0) ein Interrupt ausgelöst werden soll. Ebenfalls in diesem Register wird der Timer Prescaler, der Vorteiler für den Timer Counter eingestellt. Die folgenden Beschreibungen und die Tabelle geben Auskunft über die verschiedenen Einstellungsmöglichkeiten dieses Registers.

Register offset: \$_0D

	BIT7	6	5	4	3	2	1	BIT0
R	TOI	0	0	0	TCRE	PR2	PR1	PR0
W								
RESET:	0	0	0	0	0	0	0	0

TSCR2 Register

TOI - Timer Overflow Interrupt Enable

0: Interrupt deaktiviert

1: Beim setzen des Timer Overflow Flags wird ein Interrupt ausgelöst.

TCRE - Timer Counter Reset Enable

Wenn dieses Bit gesetzt wird, wird nach einem Output Compare am Timerkanal 7 der Timer Counter auf '0' zurückgestellt. Benötigt wird diese Funktion vor allem für die Pulsweitenmodulation. Wenn TC7 auf 0x0000 und TCRE auf '1' gesetzt ist, bleibt der Timer Counter immer auf 0x0000 stehen. Ist TC7 auf 0xFFFF und TCRE auf '1' gesetzt, zählt der Counter immer von 0x0000 bis 0xFFFF, aber das Timer Overflow Flag wird nie gesetzt werden. 0: Counter Reset Funktion ist deaktiviert, der Timer zählt immer bis zum Maximalwert. 1: Counter Reset nach Output Compare an Kanal 7. PR2, PR1, PR0 - Timer Prescaler select Über diese drei Bits wird der Vorteiler-Divisor des Timer Count Registers festgelegt. Die möglichen Konfiguration sind in der folgenden Tabelle aufgelistet:

PR2	PR1	PR0	Prescale Factor
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Der Vorteiler ist von 1 bis 128 einstellbar (TSCR2).

TFLG1 (Timer Interrupt Flag 1)

Die Bits im Interrupt Flag Register werden gesetzt, wenn sich an dem zum entsprechenden Bit gehörenden Kanal eine Interruptbedingung erfüllt hat, das Eintreffen der im TCTL3/4 festgelegten Flanke, oder ein Output Compare. Um das Flag zu löschen, muss eine '1' in das entsprechende Bit geschrieben werden, das Schreiben einer '0' hat keine Auswirkungen. Zugriffe auf dieses Register sind jederzeit möglich. Falls das TFFCA-Bit im Timer System Control Register gesetzt ist, wird bei einem Lese- oder Schreibzugriff auf das Output Compare Register (TC0 bis9) das zugehörige Bit im TFLG1 automatisch gelöscht.

Register offset: \$_0E

	BIT7	6	5	4	3	2	1	BIT0
R	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F
W	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F
RESET:	0	0	0	0	0	0	0	0

TFLG1 Register

CnF: Input Capture/Output Compare Kanal "n" Flag

TFLG2 (Timer Interrupt Flag 2)

Das Bit 7 des Timer Interrupt Flag 2 wird bei Überlauf des 16-Bit Zählers von 0xFFFF nach 0x0000 gesetzt. Um das Flag zu löschen, muss eine '1' ins Bit 7 geschrieben werden. Der Zugriff ist immer möglich. Falls im Timer System Control Register das TFFCA-Bit gesetzt ist, wird Timer Overflow Flag (Bit 7) nach jedem Zugriff auf das Timer Count Register automatisch gelöscht.

Register offset: \$_0F

	BIT7	6	5	4	3	2	1	BIT0
R	TOF	0	0	0	0	0	0	0
W	TOF							
RESET:	0	0	0	0	0	0	0	0

TFLG2 Register

TOF: Timer Overflow Flag

Weitere Informationen finden Sie unter: S12ECT16B8CV1.pdf und timer.pdf

12. Piaport

Um die Ports A und B zu testen bietet sich das Piaport E/A Modul aus dem Unterricht an. In unserem Funktionstest wird einfach ein Lauflicht ausgegeben.

Quelltext: piaport.c

13. Der Analog Digital Wandler

13.1 Allgemeines

Ein Analog/Digitalwandler setzt eine analoge Eingangsspannung in einen digitalen Ausgabewert um. Ein Eingang von 0 Volt z.B. liefert den Bitwert 0, ein Eingang von 5 Volt liefert den Bitwert 1024 (10 Bit- Auflösung, 5 Volt U_{REF} , unipolar, d.h. nur positive Eingangsspannungen). Die zu erfassende Spannung lässt sich mit analogen Schaltungen an den Analogeingang des Wandler- Bausteins anpassen.

Der MC9S12DG256 besitzt zwei integrierte Analog/Digitalwandler ATD0 und ATD1. Jeder dieser AD- Wandler ist ein 8- Kanal- 10- Bit Wandler, der nach dem Prinzip der sukzessiven Approximation mit gemultiplexten Eingängen arbeitet. Umsetzer nach dem Verfahren der Schrittweisen Näherung (Sukzessive Approximation oder Wägeverfahren) führen den Vergleich Bitweise durch. Beginnend mit der werthöchsten Bitposition wird die Vergleichsspannung probeweise um den Wert einer Bitposition erhöht, und wenn diese größer als die zu messende Spannung ist, wieder entfernt. Ein 10- Bit Wandler benötigt also zehn Schritte.

Die wichtigsten Eigenschaften sind:

- einstellbare Auflösung von 8 oder 10 Bit,
- 7 μ s Wandlungszeit für einfache 10- Bit- Wandlung,
- Mess- Zwischenverstärker für Samplevorgang,
- programmierbare Sample- Zeit,
- Ergebnisse links- oder rechtsseitig ausgerichtet,
- Ergebnisse mit oder ohne Vorzeichen
- Steuerung durch externes Triggerereignis möglich,
- Interrupt nach Beendigung der Wandlung,
- Eingangs- Multiplexer für acht Analogeingänge,
- Pins können als Analog- oder Digitaleingang gemultiplext sein,
- Wandlungssequenzen von einer bis acht Wandlungen einstellbar,
- Ununterbrochener Wandlungsmodus einstellbar,
- Mehrfachabtastungen eines Kanals möglich.

Durch Software sind verschiedene Betriebsmodi einstellbar.

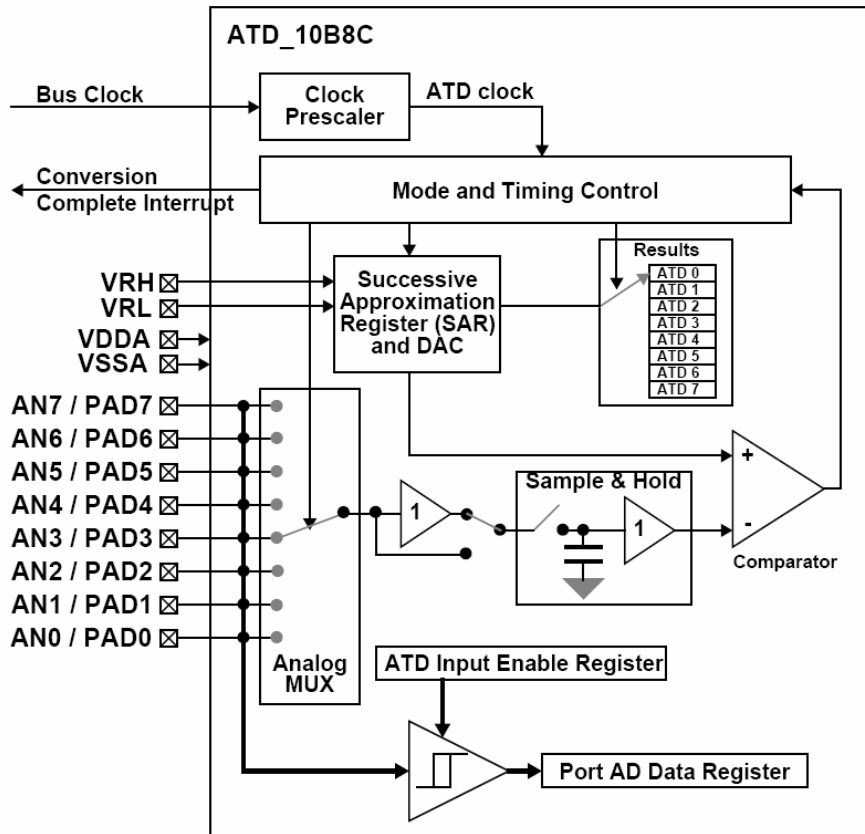


Bild 1: Blockdiagramm eines AD- Wandlers des MC9S12DG256

Der AD- Wandler hat eine Anzahl von insgesamt 12 externen Pins. Die Pins AN0 bis AN7 sind die acht analogen Eingänge des Wandlers. Alle Pins können neben der Funktion des Analog- Eingangs individuell als digitaler Port AD- Eingang konfiguriert werden. Der Pin AN7 kann zusätzlich noch als externer Triggereingang konfiguriert werden, mit dessen Hilfe der Wandlungsprozess angestoßen werden kann. Zusätzlich benutzt der AD- Wandler noch die Pins V_{RH} und V_{RL} (obere und untere Referenzspannung) sowie V_{DDA} und V_{SSA} (Spannungsversorgung für den analogen Teil des AD- Wandlers). Der AD- Wandler wird in einen analogen und in einen digitalen Subblock unterteilt. Der analoge Subblock enthält die analoge Elektronik, die benötigt wird, um eine Analog- Digitalumwandlung durchzuführen. Eine separate Analog- Spannungsversorgung wird benutzt, damit möglichst wenige Störungen vom digitalen Teil des Mikrocontrollers eingekoppelt werden.

13.2 Funktionstest/Programmfunktionen/Registerbeschreibungen

In unserem Funktionstest schließen wir einen Temperatursensor AD22100K der Firma Analog Devices an unser Modul an. Dieser erfasst Temperaturen im Bereich zwischen 0°C und 100°C . Dabei liefert er definierte Ausgangsspannungen zwischen 1,375 Volt und 3,625 Volt. Der binäre Ausgangswert des Spannungseingangswertes wird in unserem Programm in den dazugehörigen Temperaturwert gewandelt. Dieser wird in ein String geschrieben welcher aufs LCD- Display ausgegeben wird. Die erneute Ausgabe des Wertes findet nur statt, wenn sich der Temperaturwert und damit der Spannungswert am Pin ändern. Ansonsten wird weiter gewandelt und verglichen.

Quelltext: atd_test.c

Folgende spezielle Funktionen werden im Programm verwendet:

Funktion 1: **void initATD0(void)**

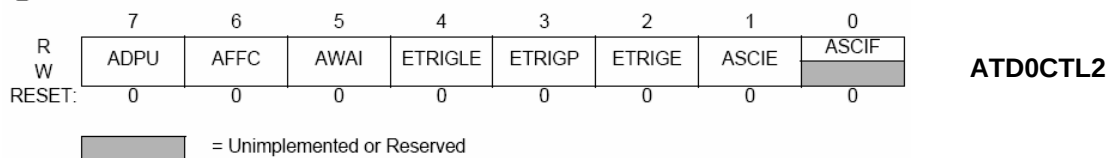
Aufgabe:	Initialisiert das ATD0 Modul (enable, 10 Bit- Auflösung, Clock Teiler = 12, 2nd sample time = 2 ATD Clock)
Header:	s12_atd.h
Funktionsaufruf:	initATD0()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	ATD0 (PAD0 – PAD7)

Registerwerte die für diese Funktion von Bedeutung sind:

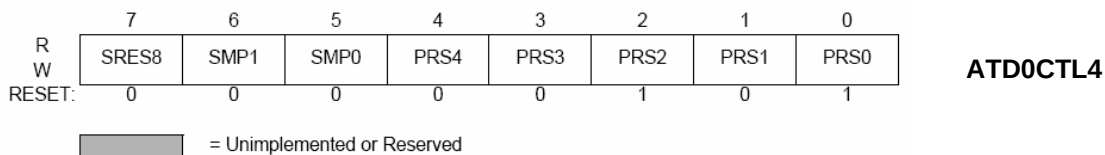
Bezeichner	Bedeutung	Werte	Effekt
ATD0CTL2	ATD0 Controlregister 2	0x80	enable ATD0
ATD0CTL4	ATD0 Controlregister 4	0x05	10 Bit Auflösung, Clock Teiler = 12

Registeraufbau:

\$_02



\$_04



$$\text{ATDclock} = \frac{[\text{BusClock}]}{[\text{PRS} + 1]} \times 0.5$$

Funktion 2: **UINT16 getATD0(UINT8 channel)**

Aufgabe:	bei Übergabe des ATD Eingangskanals, liefert die Funktion je nach Spannungswert einen digitalen Ausgangswert zwischen 0 und 1024 (10 Bit Auflösung)
Header:	s12_atd.h
Funktionsaufruf:	getATD0(UINT8 channel)
Übergabeparameter:	ATD0 Kanal (0..7)
Rückgabeparameter:	ATD0 Datenregister
Reccourcen:	ATD0 (PAD0 – PAD7)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
ATD0CTL3	ATD0 Controlregister 2	0x08	Wählt eine Umwandlung per Sequenz
ATD0CTL5	ATD0 Controlregister 4	0x8x	Rechtsbündiger Vorzeichenloser Datenmode, Arbeitet in Einzelsequence, ein Ausgang von 8 Kanälen
ATD0STAT0	ATD0 Statusregister	0x80	Sequence Complete Flag abfragen
ATD0DR0	ATD0 Ergebnisregister		Hier steht das Wandlungsergebnis

Registeraufbau:

\$_03

	7	6	5	4	3	2	1	0
R	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0
W								
RESET:	0	0	1	0	0	0	0	0

ATD0CTL3

 = Unimplemented or Reserved

\$_05

	7	6	5	4	3	2	1	0
R	DJM	DSGN	SCAN	MULT	0	CC	CB	CA
W								
RESET:	0	0	0	0	0	0	0	0

ATD0CTL5

 = Unimplemented or Reserved

\$_06

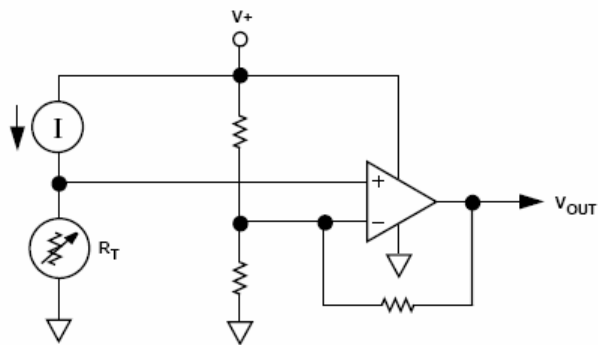
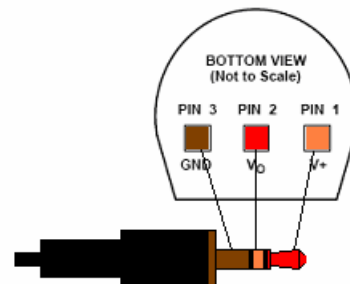
	7	6	5	4	3	2	1	0
R	SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0
W								
RESET:	0	0	0	0	0	0	0	0

ATD0STAT0

 = Unimplemented or Reserved

	7	6	5	4	3	2	1	0	
R	BIT 9 MSB	BIT 8	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	10-bit data
W	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	8-bit data
RESET:	0	0	0	0	0	0	0	0	

 = Unimplemented or Reserved

ATD0DR0**Bild 2: Temperatursensor AD22100K****Bild 3: Anschlussbelegung AD22100K**

Berechnung der Ausgangsspannung : $V_{OUT} = (V^*/5V) * [1.375V + (22.5mV/^{\circ}C) * T_a]$

Typische Ausgangsspannungen : $V^+ = 5.0V, T_a = 0^{\circ}C \rightarrow V_{OUT} = 1.375 V$
 $V^+ = 5.0V, T_a = 100^{\circ}C \rightarrow V_{OUT} = 3.625 V$

14. Digital Analog Wandler

14.1 Allgemeines

Ein Digital- Analogwandler setzt einen binären Eingangswert in eine analoge Ausgangsspannung um.

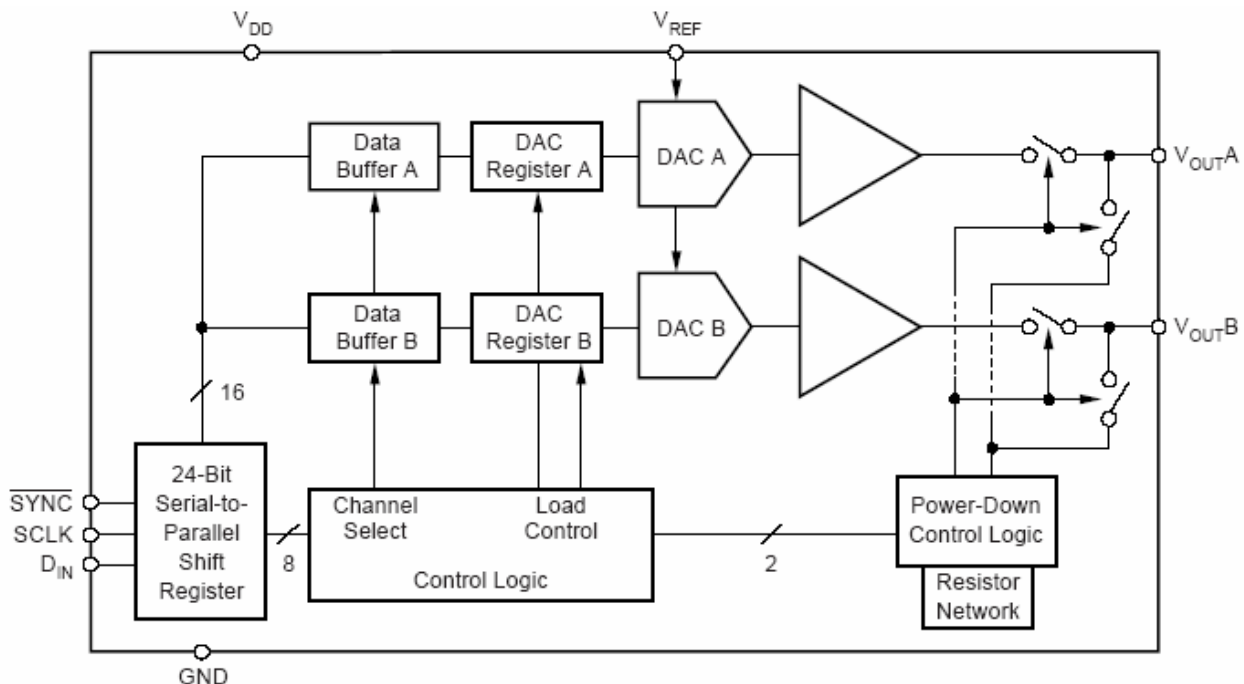


Bild 1: Blockschaltbild DAC8532

Der S12compact besitzt als weitere Schaltungsoption einen 16 Bit D/A Wandler. Zum Einsatz kommt ein DAC8532 von Burr Brown. Dieser Baustein bietet 2 Kanäle, deren Ausgangsspannungen VOUTA und VOUTB sowohl nacheinander als auch gleichzeitig aktualisiert werden können. Der IC10 bietet eine Referenzspannung (4,096V), welche die obere Ausgangsspannungsgrenze vorgibt. Sie kann durch eine externe Referenzspannungsquelle ersetzt werden, wenn zuvor die BR9 aufgetrennt wird. (Achtung: dies hat auch Auswirkung auf den optionalen ADC, der hier aber nicht implementiert ist). Der Lastwiderstand an den Ausgängen sollte 2kOhm nicht unterschreiten. Beim Einschalten der Betriebsspannung wird die Ausgangsspannung der beiden DAC- Kanäle im IC auf 0V zurückgesetzt. Der DAC wird – wie auch RTC, ADC, und die Zusatz- I/Os – über SPI0 angesteuert. /SPICS1 dient als Chipselectsignal für den DAC. Die Ausgangssignale VOUTA und VOUTB sind als gesicherte Ausgänge in Form eines Klinkensteckers verfügbar.

14.2 Funktionstest/Programmfunktionen/Registerbeschreibungen

In unserem Funktionstest fordern wir über das Hyperterminal, zur Eingabe des DAC- Kanals (DACA, DACB) auf. Danach wird die gewünschte Spannung an diesem Kanal eingegeben. In unserem Programm setzen wir dann den eingegebenen Wert in den 16- Bit Wert um, der dieser Spannung entspricht. Da am DAC eine Referenzspannung von 4,096V (FS) anliegt, ist dieser bei unipolar Betrieb der größte 16- Bit Wert (0xffff) zugeordnet. Daraus ergibt sich ebenfalls eine Auflösung von 62.5µV/Bit (U_{LSB}).

Quelltext: DA_test.c

Folgende spezielle Funktionen werden im Programm verwendet:

Funktion 1: **void putdac(UINT8 channel, UINT16 value)**

Aufgabe:	SPI Setup- Mode wird durchgeführt, Kanal DACA oder DACB wird ausgewählt, 16 Bit Wert wird über die SPI Schnittstelle an den DAC gesendet.
Header:	DA.h
Funktionsaufruf:	putdac(UINT8 channel, UINT16 value)
Übergabeparameter:	channel („A“ für DACA, „B“ für DACB)
Rückgabeparameter:	keine
Reccourcen:	PortH: PH4 – PH6 (für DAC enable) PortS: MOSI (serielle Datenübertragung) , SCK (Taktsynchronis.)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRH	PortH Datenrichtungsregister	0xF0	0 - 3 Eingang, 4 - 7 Ausgang
PTH	PORTH (PIM)	0x10	D/A Wandler Enable
SPI0CR1	SPI Control Register 1	0x54	SPI Enable, CPOL=0; CPHA=1

Registeraufbau:

\$0262	DDRH	Read: Write:	DDRH7	DDRH7	DDRH5	DDRH4	DDRH3	DDRH2	DDRH1	DDRH0	DDRH
--------	------	-----------------	-------	-------	-------	-------	-------	-------	-------	-------	-------------

\$0260	PTH	Read: Write:	PTH7	PTH6	PTH5	PTH4	PTH3	PTH2	PTH1	PTH0	PTH
--------	-----	-----------------	------	------	------	------	------	------	------	------	------------

	Bit 7	6	5	4	3	2	1	Bit 0	
R	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE	SPI0CR1
W									
Reset:	0	0	0	0	0	1	0	0	

Aufgerufene Unterfunktionen:

Unterfunktion 1: **UINT8 xferSPI0(UINT8 abyte)**

Aufgabe:	Senden eines Bytes über die SPI Schnittstelle
Header:	s12_spi.h
Funktionsaufruf:	xferSPI0(UINT8 abyte)
Übergabeparameter:	UINT8 abyte (Byte was über die SPI geschickt werden soll)
Rückgabeparameter:	Inhalt des SPI Datenregisters
Reccourcen:	PortS: MISO, MOSI (serielle Datenübertragung), SCK (Taktsynchronisierung)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
SPI0DR	SPI Datenregister		Byte was über die SPI geschickt werden soll
SPI0SR	SPI Statusregister	0x80	Abfragen ob SPI beschäftigt ist

Registeraufbau:

	Bit 7	6	5	4	3	2	1	Bit 0
R								
W								
Reset:	0	0	0	0	0	0	0	0

SPI0DR

	Bit 7	6	5	4	3	2	1	Bit 0
R	SPIF	0	SPTEF	MODF	0	0	0	0
W								
Reset:	0	0	1	0	0	0	0	0

SPI0SR

= Reserved

15. Pulsweitenmodulationsausgänge

15.1 Allgemeines

PWM- Ausgänge geben ein Rechtecksignal aus, Frequenz und Tastverhältnis lassen sich während des Betriebes ändern. Die Funktion kann periodische Interrupts auslösen.

Der im MC9S12DG256 genutzte PWM- Block hat 8 Ausgabekanäle, bei denen unabhängig eingestellt werden kann, ob das Signal zentriert oder linksbündig ausgegeben werden soll. Jeder der 8 Kanäle hat eine frei einstellbare Periode und Pulsweite, sowie einen speziellen Zähler.

Die Eigenschaften des PWM- Blocks sind folgende:

- acht unabhängige PWM- Kanäle mit programmierbarer Periode und Pulsweite,
- spezieller Zähler für jeden PWM- Kanal,
- programmierbare PWM- Aktivierung oder Deaktivierung für jeden Kanal,
- Periode und Pulsweite werden über doppelt gepufferte Register eingestellt. Änderungen werden erst mit dem Anfang jeder neuen Periode übernommen.
- einstellbare zentrierte oder linksbündig ausgerichtete Ausgabekanäle,
- alternativ wählbare acht 8-Bit-Kanäle oder vier Kanäle mit 16-Bit-Auflösung,
- vier wählbare Taktgeberquellen für einen breiten Frequenzbereich,
- programmierbare Takt-Selektions-Logik,
- Notabschaltungsfunktion

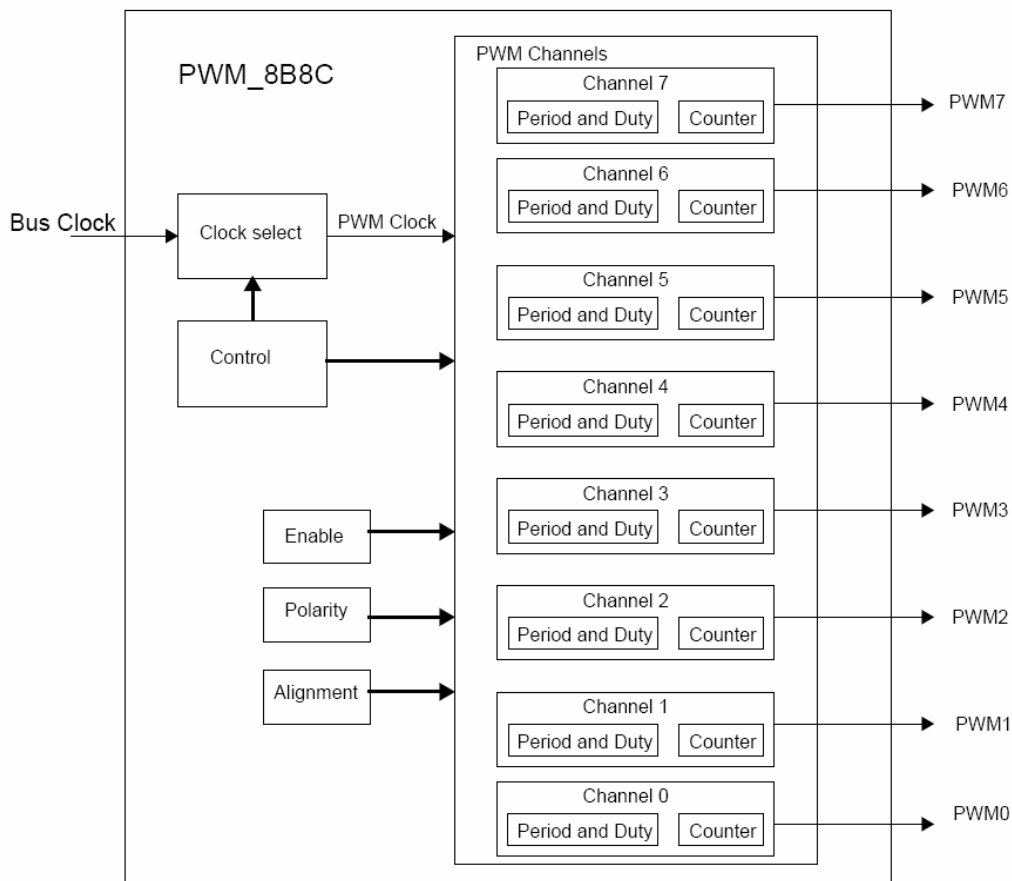


Bild 1: Blockschaltbild PWM-Modul

15.2 Funktionstest/Programmfunktionen/Registerbeschreibungen

In unserem Funktionstest fordern wir über das Hyperterminal zur Auswahl eines Alarmsignals auf. Ist dieses eingegeben, schrillt es aus einem Lautsprecher welcher am PP0 Ausgang angeschlossen ist. Die Frequenzen für die Alarmsignale werden natürlich im PWM-Modul erzeugt.

Quelltext: PWM_test.c

Folgende spezielle Funktionen werden im Programm verwendet:

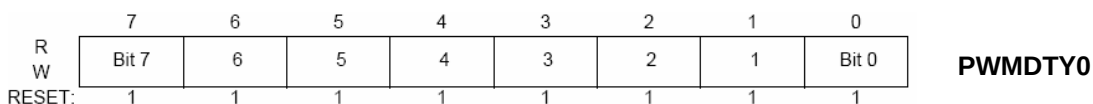
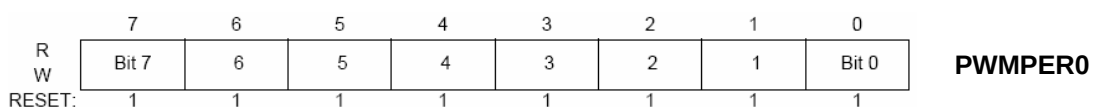
Funktion 1: **void Warnsignal1 (void)**

Aufgabe:	Warnsignal 1: 3 kurze Töne
Header:	PWM_Warnsignale.h
Funktionsaufruf:	Warnsignal1()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PORTP (PP0)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PWMPER0	PWM Periodenregister		PWM Periode
PWMDTY0	PWM Dienstregister		PWM Pulsweite

Registeraufbau:



Aufgerufene Unterfunktionen:

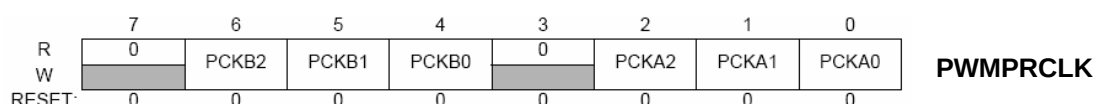
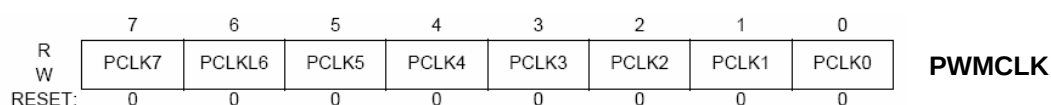
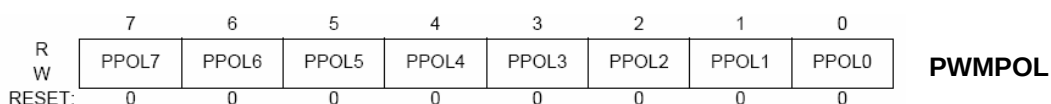
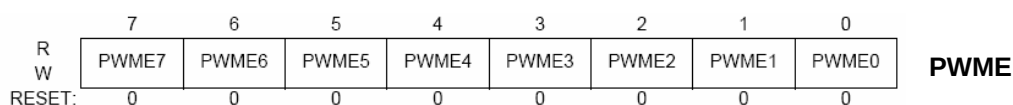
Unterfunktion 1: **void PWM_ini (void)**

Aufgabe:	Initialisierung, PWM PP0 enable, Takt startet High, Clock B ist die Clock Quelle, Clock B ist der Bus Takt, links ausgerichteter Ausgangsmode, 8-bit PWM Kanäle
Header:	PWM_Warnsignale.h
Funktionsaufruf:	PWM_ini (void)
Übergabeparameter:	Keine
Rückgabeparameter:	Keine
Reccourcen:	PORTP (PP0)

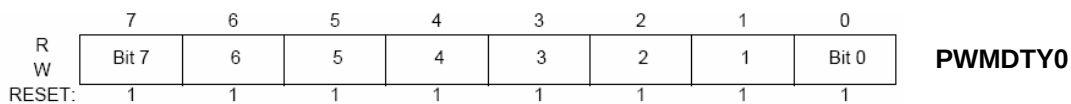
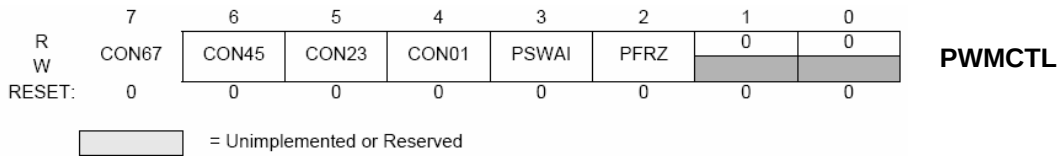
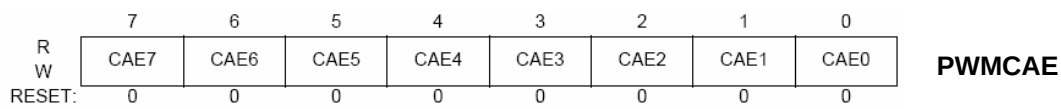
Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PWME	PWM Enable Register	0x01	PWM PP0 enable
PWMPOL	PWM Polarity Register	0xff	Takt startet High
PWMCLK	Clock Select Register	0x00	Clock B ist die Clock Quelle
PWMPRCLK	Prescale Clock Select Register	0x00	Clock B ist der Bus Takt, keine Teilung
PWMCAE	Center Align Enable Register	0x00	Links ausgerichteter Ausgangsmode
PWMCTL	PWM Control Register	0x00	8-bit PWM Kanäle
PWMDTY0	PWM Channel 0 Duty Register	0x80	Takt initialisiert auf 0x80

Registeraufbau:



= Unimplemented or Reserved



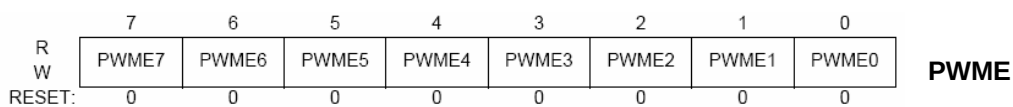
Unterfunktion 2: **void PWM_disable (void)**

Aufgabe:	PWM an PP0 wird abgeschalten
Header:	PWM_Warnsignale.h
Funktionsaufruf:	PWM_disable();
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PORTP (PP0)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PWME	PWM Enable Register	0xFE	PWM PP0 disable

Registeraufbau:



Unterfunktion 3: **void Delay0(int delayTime)**

Aufgabe:	Kurze Warteschleife
Header:	delay.h
Funktionsaufruf:	Delay0(int delayTime);
Übergabeparameter:	Integerwert 0 – 0xFFFF
Rückgabeparameter:	keine

Unterfunktion 4: **void Delay1(int delayTime)**

Aufgabe:	Lange Warteschleife
Header:	delay.h
Funktionsaufruf:	Delay1(int delayTime);
Übergabeparameter:	Integerwert 0 – 0xFFFF
Rückgabeparameter:	keine

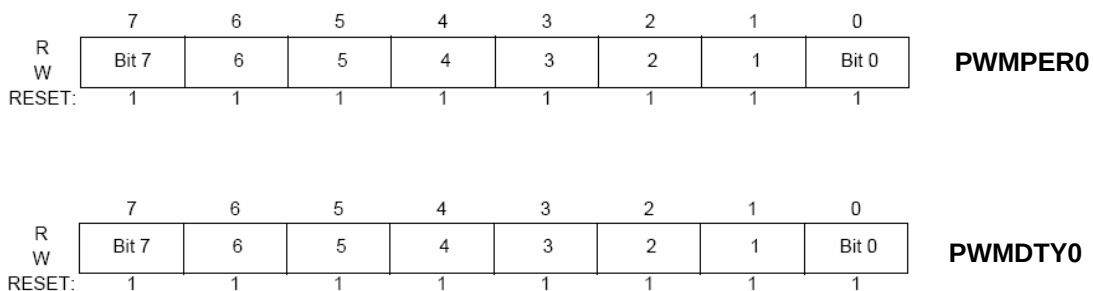
Funktion2: **void Warnsignal2 (void)**

Aufgabe:	Warnsignal 2: 3 mal Glocke
Header:	PWM_Warnsignale.h
Funktionsaufruf:	Warnsignal2()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PORTP (PP0)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PWMPER0	PWM Periodenregister		PWM Periode
PWMDTY0	PWM Dienstregister		PWM Pulsweite

Registeraufbau:



Aufgerufene Unterfunktionen:

Unterfunktion 1: **void PWM_ini (void)**

Unterfunktion 2: **void PWM_disable (void)**

Unterfunktion 3: **void Delay0 (int delayTime)**

Unterfunktion 4: **void Delay1 (int delayTime)**

Siehe PWM Warnsignale 1!

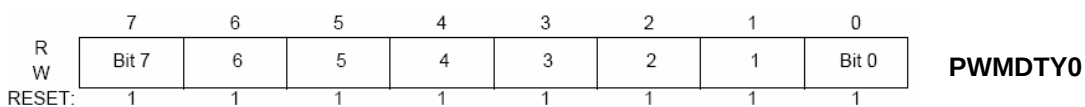
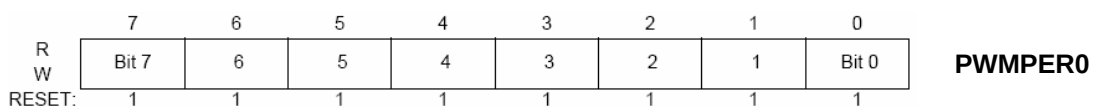
Funktion3: **void Warnsignal3 (void)**

Aufgabe:	Warnsignal 3: 3 mal Telefon
Header:	PWM_Warnsignale.h
Funktionsaufruf:	Warnsignal3()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PORTP (PP0)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PWMPER0	PWM Periodenregister		PWM Periode
PWMDTY0	PWM Dienstregister		PWM Pulsweite

Registeraufbau:



Aufgerufene Unterfunktionen:

Unterfunktion 1: **void PWM_ini (void)**

Unterfunktion 2: **void PWM_disable (void)**

Unterfunktion 4: **void Delay1 (int delayTime)**

Siehe PWM Warnsignale 1!

16. Keypad

16.1 Allgemeines

Das Keypad (16 Tasten Hexadezimal (RAFI)) besitzt außer der Spannungsversorgung 5 Anschlüsse. DA (DATA available) zeigt den Status des Keypads an. 0 bedeutet keine Taste betätigt, 1 bedeutet Taste betätigt. Die Daten an den Pins 1 - 4 sind nur gültig wenn DA eine 1 anzeigt. Im Polling Betrieb wird DA regelmäßig abgefragt. Außerdem kann DA als Eingang für einen Interrupt benutzt werden.

Folgende Tastencodes liegen bei Betätigung der Tasten an:

Taste:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Code (hex):	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

Diese Codes werden in einer Funktion in den dazugehörigen Tastewert gewandelt.

16.2 Funktionstest/Programmfunktionen/Registerbeschreibungen

Für das Keypad- Modul führten wir zwei Funktionstests durch. Einmal im Polling- Betrieb, wo ständig das DA Bit abgefragt wird und einmal im Interruptbetrieb, wo bei einer Tastenbetätigung ein Interrupt ausgelöst wird. Der Interruptbetrieb hat gegenüber dem Pollingbetrieb zum Vorteil, dass keine Rechenzeit der CPU gebunden wird, es kann nach Prioritäten geordnet werden und mehrere Programme können quasi gleichzeitig ablaufen. Der Pollingbetrieb ist dagegen einfacher zu programmieren.

Im 1. Beispiel wird das Keypad im Pollingbetrieb verwendet. Das Keypad wird abgefragt und bei Betätigung einer Taste, wird der Wert auf dem LCD Display ausgegeben.

Quelltext: Keypad_Lcd_poll.c

Folgende spezielle Funktionen werden im Programm verwendet:

Funktion 1: **UINT8 keypad_poll (void)**

Aufgabe:	fragt Keypad im Pollingbetrieb ab und liefert als Rückgabewert den fertig gewandelten Tastencode
Header:	keypad.h
Funktionsaufruf:	keypad_poll()
Übergabeparameter:	keine
Rückgabeparameter:	Wert der gedrückten Taste
Reccourcen:	PortP (PP2...PP6)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PTIP	PortP Inputregister		Gibt PORTP immer als Eingang wieder

Registeraufbau:

	Bit 7	6	5	4	3	2	1	Bit 0	
Read:	PTIP7	PTIP6	PTIP5	PTIP4	PTIP3	PTIP2	PTIP1	PTIP0	PTIP
Write:									
Reset:	-	-	-	-	-	-	-	-	

 = Reserved or unimplemented

Im 2. Beispiel wird das Keypad im Interruptbetrieb betrieben. Während ein Programm abläuft, in diesem Fall unsere Laufschrift aus dem LCD Test, kann eine Taste betätigt werden, womit ein Interrupt ausgelöst wird. Da das Keypad an Port P angeschlossen ist, wird in der Interruptvektortabelle in den Vektor von Port P, die Startadresse der Interruptservice-routine gelegt (Port_P_ISR). Zur Interruptfreigabe muss der Pin 6 (DA) an Port P zum Interrupt freigegeben werden. Sobald eine Taste betätigt wird, wird die Interruptserviceroutine abgearbeitet. In dieser wird der Tastecode eingelesen, gewandelt und an die Adresse 3000 in den RAM des HCS12 Compact geschrieben.

Quelltext: Keypad_Lcd_int.c

Folgende spezielle Funktionen werden im Programm verwendet:

Funktion 1: **void keypad_int (void)**

Aufgabe:	gibt Interrupts Frei, Interruptenablebit an PP6 wird gesetzt, Interrupt reagiert auf fallende Flanke
Header:	keypad.h
Funktionsaufruf:	keypad_int()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PortP (PP2...PP6)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PIEP	PORT P Interrupt Enable Register	0x40	PP6 Interrupt Enable
PPSP	PORT P Polarity Select Register	0xBF	Interrupt reagiert auf fallende Flanke
DDRP	PORT P Data Direction Register	0xBF	PP6 auf Eingang

Registeraufbau:

	Bit 7	6	5	4	3	2	1	Bit 0	
Read:	PIEP7	PIEP6	PIEP5	PIEP4	PIEP3	PIEP2	PIEP1	PIEP0	PIEP
Write:									
Reset:	0	0	0	0	0	0	0	0	

	Bit 7	6	5	4	3	2	1	Bit 0	
Read:	PPSP7	PPSP6	PPSP5	PPSP4	PPSP3	PPSP2	PPSP1	PPSP0	PPSP
Write:									
Reset:	0	0	0	0	0	0	0	0	

	Bit 7	6	5	4	3	2	1	Bit 0	
Read:	DDRP7	DDRP6	DDRP5	DDRP4	DDRP3	DDRP2	DDRP1	DDRP0	DDRP
Write:									
Reset:	0	0	0	0	0	0	0	0	

Interruptserviceroutine 1: `interrupt void Port_P_ISR (void)`

Aufgabe:	Interruptserviceroutine, Wandlung des Tastencodes und schreiben an die Adresse 3000 in den RAM
Header:	keypad.h
Funktionsaufruf:	keypad_int()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PortP (PP2...PP6)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PIFP	PORT P Interrupt Flag Register	0x40	Interruptflag zurücksetzen
PTIP	PortP Inputregister		Gibt PORTP immer als Eingang wieder

Registeraufbau:

	Bit 7	6	5	4	3	2	1	Bit 0	
Read:	PIFP7	PIFP6	PIFP5	PIFP4	PIFP3	PIFP2	PIFP1	PIFP0	PIFP
Write:									
Reset:	0	0	0	0	0	0	0	0	

	Bit 7	6	5	4	3	2	1	Bit 0	
Read:	PTIP7	PTIP6	PTIP5	PTIP4	PTIP3	PTIP2	PTIP1	PTIP0	PTIP
Write:									
Reset:	-	-	-	-	-	-	-	-	

 = Reserved or unimplemented

17. LCD Display

17.1 Allgemeines

Als LCD Display kommt hier ein 2x16 Punkt Matrix Display zum Einsatz (2 Zeilen, 16 Zeichen). Das einheitliche Verhalten vieler Displays verdanken wir einem Controllerchip, der sich als Standard durchgesetzt hat, und der von vielen Herstellern verbaut wird. Dabei handelt es sich um den HD44780 von Hitachi, der hier zum Einsatz kommt.

Dieser Chip besitzt

- einen 80 Zeichen langen Textpuffer
- ein paralleles Interface
- LCD Treiber mit 16 Punkt-Zeilenleitungen und 40 Punkt-Spaltenleitungen.

Das Dotmatrix-Display wird mit einer Betriebsspannung V_{dd} von +5V ($\pm 5\%$) betrieben. Der V_{ss} - Pin ist der Masseanschluss. Die Stromaufnahme (ohne Beleuchtung) liegt meist unter 1 mA. Maximalwerte von 5 mA sind möglich. Außerdem wird eine Spannung zur Einstellung des Displaykontrastes benötigt.

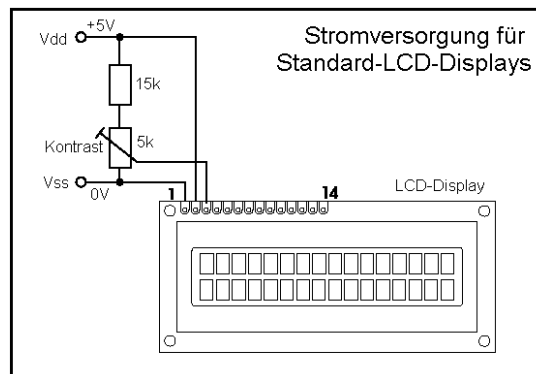


Bild 1: Stromversorgung

Der Controller (und damit die LCD-Anzeige) besitzt ein paralleles Interface, bestehend aus:

- ein 8 Bit Datenbus (D0..D7)
- Enable-Leitung
- RS-Leitung
- R/W-Leitung

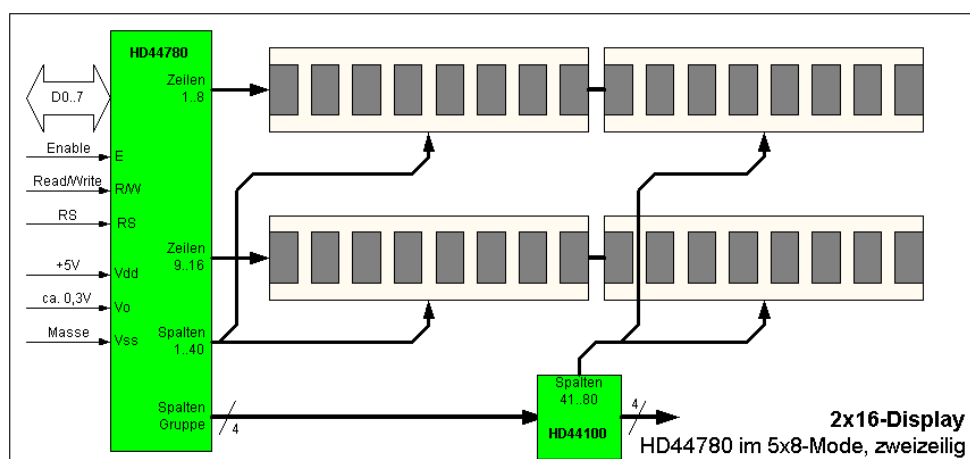


Bild 2: 2x16-Display, HD44780 im 5x8-Mode, zweizeilig

Es gibt zwei verschiedene Möglichkeiten zur Ansteuerung eines solchen Displays, den 8Bit- und den 4Bit- Modus. Für den 8Bit-Modus werden, wie der Name schon sagt, alle acht Datenleitungen zur Ansteuerung verwendet. Somit kann durch einen Zugriff immer ein ganzes Byte übertragen werden. Der 4Bit- Modus verwendet nur die oberen vier Datenleitungen (DB4-DB7). Um ein Byte zu übertragen braucht man deshalb zwei Zugriffe, wobei zuerst das höherwertige "Nibble" (=4 Bits), also Bit 4 bis Bit 7 übertragen wird, und dann das niederwertige, also Bit 0 bis Bit 3. Die unteren Datenleitungen des LCDs werden einfach auf GND gelegt.

Der 4Bit-Modus hat den Vorteil, dass man 4 IO-Pins weniger benötigt als beim 8Bit-Modus, weshalb wir uns hier für eine Ansteuerung mit 4Bit entschieden haben.

Neben den vier Datenleitungen (DB4, DB5, DB6 und DB7) werden noch die Anschlüsse RS, RW und E benötigt.

Über RS wird ausgewählt, ob man einen Befehl oder ein Datenbyte an das LCD schicken möchte. Ist RS Low, dann wird das ankommende Byte als Befehl interpretiert, ist RS High, dann wird das Byte auf dem LCD angezeigt.

RW legt fest, ob geschrieben oder gelesen werden soll. High bedeutet lesen, Low bedeutet schreiben. Wenn man RW auf lesen einstellt und RS auf Befehl, dann kann man das Busy-Flag lesen, das anzeigt ob das LCD den vorhergehenden Befehl fertig verarbeitet hat. Ist RS auf Daten eingestellt, dann kann man z.B. den Inhalt des Displays lesen.

LCD Steuerung:

RS	R/W	Funktion
0	0	Schreiben ins Steuerregister IR
0	1	Lese Busy-Flag und Adresszähler
1	0	Schreiben ins Datenregister und Display Data RAM
1	1	Lese Daten vom Display Data RAM

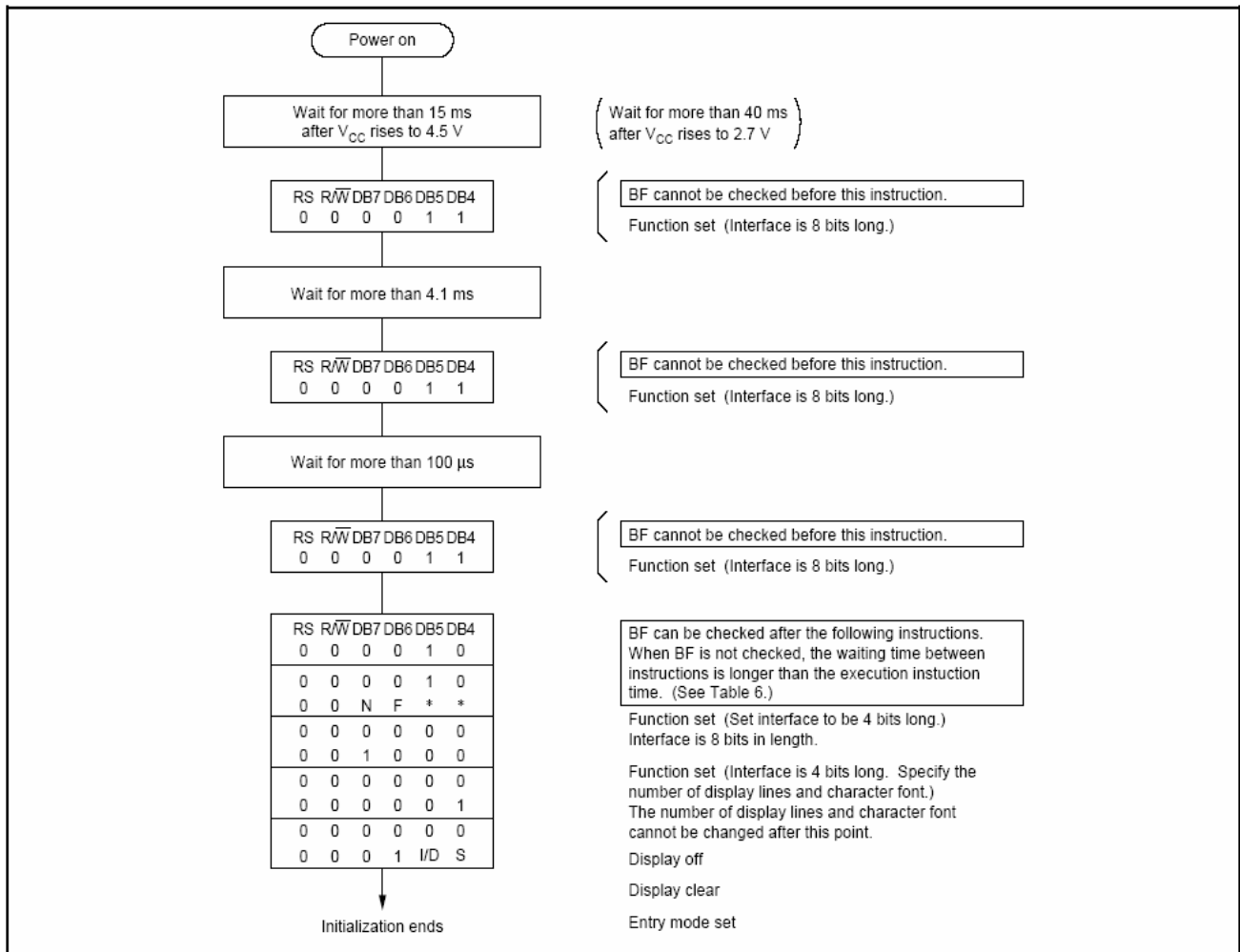


Bild 3: Programmablaufplan 4 Bit- Initialisierung

17.2 Funktionstest/Programmfunktionen/Registerbeschreibungen

In unserem Funktionstest lassen wir eine Laufschrift mit dem Titel: „Projektarbeit HCS12 Compact“ über das Display laufen.

Quelltext: LCD_Test.c

Folgende spezielle Funktionen werden im Programm verwendet:

Funktion 1: **void Lcd_ini (void)**

Aufgabe:	Initialisierung der LCD Anzeige (Function set, 4-Bit Interface, zwei-zeilig, 5:7 Punktmatrix, Display ein, Cursor aus, Blinken aus, Anzeige löschen, Adresszähler 00, Entry Mode, Adresszähler autoincrement)
Header:	Lcd.h
Funktionsaufruf:	Lcd_ini()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PortK (PK0...PK7)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRK	PORTK Datenrichtungsregister		Datenrichtungsregister
PORTK	PORTK		Wert an PORTK

Registeraufbau:

Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
PORTK	Read:	Bit 7	6	5	4	3	2	1	Bit 0	PORTK
	Write:									
DDRK	Read:	Bit 7	6	5	4	3	2	1	Bit 0	DDRK
	Write:									

Aufgerufene Unterfunktion:

Unterfunktion 1: **void delay16 (UINT16 i)**

Aufgabe:	Integerwert wird runtergezählt
Header:	delay.h
Funktionsaufruf:	delay16();
Übergabeparameter:	Integerwert 0 – 0xFFFF
Rückgabeparameter:	keine

Funktion 2: **void s_steuer (UINT8 s)**

Aufgabe:	dient zur Steuerung des Displays, Steuerbyte wird in Nibble zerlegt und nacheinander ins Steuerregister geschrieben
Header:	Lcd.h
Funktionsaufruf:	s_steuer (UINT8 s)
Übergabeparameter:	Steuerbyte z.B. 0x00 → Cursor 1. Zeile, 1. Zeichen 0x40 → Cursor 2. Zeile, 1. Zeichen
Rückgabeparameter:	keine
Reccourcen:	PortK (PK0...PK7)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRK	PORTK Datenrichtungsregister		Datenrichtungsregister
PORTK	PORTK		Wert an PORTK

Registeraufbau:

[illegible]

Funktion 3: **void s_data (UINT8 s)**

Aufgabe:	Datenbyte wird in Nibble zerlegt, diese werden nacheinander ins Datenregister geschrieben und damit auf dem Display dargestellt
Header:	Lcd.h
Funktionsaufruf:	s_data (UINT8 s)
Übergabeparameter:	Datenbyte s (ASCII's die auf dem Display dargestellt werden sollen)
Rückgabeparameter:	keine
Reccourcen:	PortK (PK0...PK7)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
DDRK	PORTK Datenrichtungsregister		Datenrichtungsregister
PORTK	PORTK		Wert an PORTK

Registeraufbau:

Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
PORTK	Read:	Bit 7	6	5	4	3	2	1	Bit 0	PORTK
	Write:									
DDRK	Read:	Bit 7	6	5	4	3	2	1	Bit 0	DDRK
	Write:									

Aufgerufene Unterfunktionen für s_data() und s_steuer():

Unterfunktion 1: **void enable (void)**

Aufgabe:	Enable LCD (Port K kurz auf enable und wieder disable)
Header:	Lcd.h
Funktionsaufruf:	enable ()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PortK (PK0...PK7)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PORTK	PORTK		Wert an PORTK

Registeraufbau:

Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
PORTK	Read:	Bit 7	6	5	4	3	2	1	Bit 0	PORTK
	Write:									
DDRK	Read:	Bit 7	6	5	4	3	2	1	Bit 0	DDRK
	Write:									

Unterfunktion 2: **void I_busy (void)**

Aufgabe:	Abfrage des Busy Flags
Header:	Lcd.h
Funktionsaufruf:	I_busy ()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PortK (PK0...PK7)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
PORTK	PORTK		Wert an PORTK

Registeraufbau:

Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
PORTK	Read:	Bit 7	6	5	4	3	2	1	Bit 0	PORTK
	Write:									
DDRK	Read:	Bit 7	6	5	4	3	2	1	Bit 0	DDRK
	Write:									

18. CAN – Schnittstelle

18.1 Überblick

CAN ist die Abkürzung für Controller Area Network und bezeichnet eine gemultiplexte serielle Kommunikation bis zu einer Datenübertragungsrate von 1 MBit pro Sekunde. Die Kommunikation findet zwischen den verschiedenen Knoten (Nodes) statt, die alle an einem zweiadrigen Kabel verbunden sind. Die Kommunikation findet unsynchronisiert zwischen den Busteilnehmern statt, d.h. es wird kein Synchronisationstakt mit übertragen, und die Synchronisation wird über die Daten selbst vorgenommen. Alle Stationen können „jederzeit“ senden, und alle Stationen empfangen alle Botschaften auf dem CAN-Bus (Multi-Master). CAN ist ein Standard in der Automobilindustrie und wird auch vielfach in Industrieanwendungen verwendet.

Der MC9S12DG256 verfügt über zwei CAN-Module, die mit CAN0 und CAN4 bezeichnet werden. CAN0 kommuniziert über die Portpins PM0 und PM1 mit einem On-Board CAN-Interface Chip (IC5), welcher das physische Businterface bildet. Wenn der S12compact der letzte Knoten am CAN-Bus ist, wird eine Terminierung erforderlich. Sie kann durch schließen der Lötbrücke BR2 aktiviert werden. Das Businterface für den CAN4 ist nicht auf dem S12compact realisiert. Dieses haben wir auf unserer Trägerplatine durch den CAN-Interface Chip PCA82C250T realisiert. Die TTL Signale für CAN4 werden über die Portpins PM6 und PM7 geleitet.

18.2 msCAN-Implementation des HCS12-Mikrocontrollers

Bei CAN können Daten zu jeder Zeit und in beliebiger Menge von jeder Station gesendet und empfangen werden. Da der Mikrocontroller nur eine begrenzte Leistungsfähigkeit zur Verfügung stellt, die empfangenen Daten zu verarbeiten, werden sie in Puffern zwischengespeichert. Zum Senden und Empfangen von Daten stehen auf dem msCAN-Modul getrennte Puffer zur Verfügung. Fünf Empfangspuffer (Rx-Puffer) sorgen dafür, dass alle empfangenen Daten von der CPU verarbeitet werden können, ohne die CPU zu sehr zu beanspruchen. Die Sendedaten können auf drei Sendepuffer aufgeteilt werden, wobei jeder Puffer mit einer Priorität versehen werden kann. Das msCAN-Modul sendet dann denjenigen Puffer mit der höchsten Priorität zuerst, wenn mehrere gefüllte und sendebereite Puffer zur Verfügung stehen.

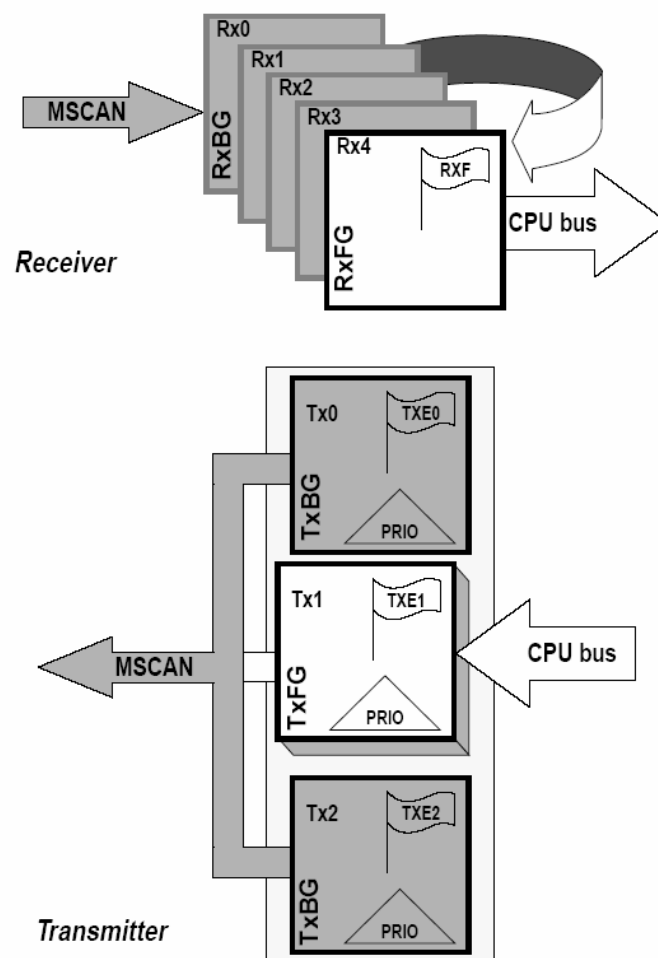


Bild 1: Puffer-Schema des msCAN-Moduls

18.3 Sendevorgang des msCAN-Modul

Heutzutage setzt manche Software der Anwendungsschicht voraus, dass eine CAN-Station (Node) einen kontinuierlichen Datenstrom, bestehend aus einer bestimmten Reihenfolge von Botschaften, senden und empfangen kann. Damit man die Daten lückenlos versenden kann stellt das msCAN- Modul beim MC9S12DG256 drei Sendepuffer zur Verfügung. Dann kann die CPU einen Sendepuffer füllen, während das msCAN-Modul einen anderen Puffer sendet. Jedem zum Senden bereiten Puffer kann eine Priorität von 0x00 bis 0xFF zugeteilt werden, die die Reihenfolge des Sendens vorgibt. Jeder Sendepuffer besteht aus 13 Byte, die den Identifier (4 Byte), 8 Bytes Daten und das Datenlängenregister enthalten. Zusätzlich steht noch ein Byte für die Priorität und ein 16-Bit-Wort für die Zeitmarke zur Verfügung, die aber nicht mit übertragen werden.

Ein Sendevorgang besteht aus vier Schritten:

1. Identifizieren eines verfügbaren leeren Sendepuffers
2. Setzen eines Zeigers auf diesen Puffer
3. Beschreiben des Sendepuffers (Identifier, Steuerbit, Daten)
4. Markieren des Sendepuffers als Sendebereit

Aufbau des Sendepuffers beim msCAN-Modul:

Addr	Register Name
\$_x0	Identifier Register 0
\$_x1	Identifier Register 1
\$_x2	Identifier Register 2
\$_x3	Identifier Register 3
\$_x4	Data Segment Register 0
\$_x5	Data Segment Register 1
\$_x6	Data Segment Register 2
\$_x7	Data Segment Register 3
\$_x8	Data Segment Register 4
\$_x9	Data Segment Register 5
\$_xA	Data Segment Register 6
\$_xB	Data Segment Register 7
\$_xC	Data Length Register
\$_xD	Transmit Buffer Priority Register ¹
\$_xE	Time Stamp Register (High Byte) ²
\$_xF	Time Stamp Register (Low Byte) ³

18.4 Empfangsvorgang beim msCAN-Modul

Für den Empfangsvorgang stehen fünf Empfangspuffer zur Verfügung. Die CPU kann jeweils nur auf einen Puffer dem Vordergrundpuffer zugreifen. Die anderen Puffer sind aus dem Speicherbereich ausgeblendet. Alle Empfangspuffer enthalten jeweils 15 Byte einschließlich Steuerungsbits, Identifiers, Datenbytes und Zeitmarke. Das RXF (Receiver Full Flag) im Receiver Flag Register zeigt den Zustand des Vordergrundpuffers an. Es ist gesetzt, wenn sich eine korrekt empfangene Botschaft im Vordergrundpuffer befindet. Ist das der Fall, kann die CPU auf den Puffer zugreifen und die Botschaft auswerten. Jede auf dem CAN-Bus befindliche Botschaft wird in den Hintergrundpuffer geschrieben und abhängig von dem Filter (Identifier Acceptance Filter) wird die CPU benachrichtigt. Damit verhindert man, dass Botschaften, die nicht für die Station bestimmt sind, die CPU belasten.

18.5 Filtern von Botschaften im msCAN

Beim msCAN Modul stehen acht Akzeptanzfilterregister (msCAN Identifier Acceptance register, CANIDAR0 bis CANIDAR7) zur Verfügung, deren Inhalt mit jedem ankommenden Identifier verglichen wird. Es werden nur solche Botschaften an die CPU gemeldet, bei denen der Akzeptanzfilter identisch mit dem Identifier der Botschaft ist. Es gibt zwei 32 Bit breite Akzeptanzfilter, die jeweils einen 29-Bit-Inhalt für das Extended-Format, vier 11-Bit – Inhalte für das Standardformat oder acht 8-Bit-Inhalte enthalten können. Für das Extended Format könnten nur zwei unterschiedliche Botschaften herausgefiltert werden, das wäre für praktische Anwendungen zu wenig. Deshalb hat das msCAN-Modul noch ein Maskenregister (msCAN Identifier Mask register, CANIDMR0 bis CANIDMR7), in dem jedes entsprechende Bit bestimmt, ob ein Bit im Maskenregister gesetzt (AM-Bit=1), wird das Bit an der Stelle im Akzeptanzfilter ignoriert. Das ermöglicht ganze Gruppen von Botschaften herauszufiltern.

Filterung von Extended (29 Bit) Identifiern

Übergabewert	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Akzeptanzfilter 1	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
Akzeptanzfilter 2	ID20	ID19	ID18	SRR = 1	IDE = 1	ID17	ID16	ID15
Akzeptanzfilter 3	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7
Akzeptanzfilter 4	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR
Maskenregister 1	M	M	M	M	M	M	M	M
Maskenregister 2	M	M	M	M	M	M	M	M
Maskenregister 3	M	M	M	M	M	M	M	M
Maskenregister 4	M	M	M	M	M	M	M	M

M = 0 ID Bit muss mit Code übereinstimmen

M = 1 ID Bit ist für die Filterung nicht relevant

Um 29 Bit Identifier Filtern zu können, müssen die Bits 19 (IDE) und 20 (SRR) gesetzt sein

18.6 Taktversorgung des msCAN Moduls

Das msCAN Modul kann Baudraten von 10 kbps bis 1 Mbps erzeugen. Dazu verwendet es entweder als Taktquelle den IP-Bus Takt oder den OSCCLK-Takt aus dem CRG-(Clock and Reset Generator) Modul. Um die verschiedenen Taktraten zu erzeugen, kann der Takt im msCAN Modul noch einmal heruntergeteilt werden. Dazu ist ein Vorteiler vorhanden, der den CANCLK-Takt mit dem Faktor 1 bis 64 teilen kann. Der resultierende Takt, der Time-Quanta-Takt Tq ist die kleinste Takteinheit im msCAN Modul. Aus diesem Takt werden die Bitzeiten für die Übertragung erzeugt. Die kleinste Informationseinheit, das Bit, wird also in weitere Segmente unterteilt, deren Grundeinheit das Zeitquantum Tq ist.

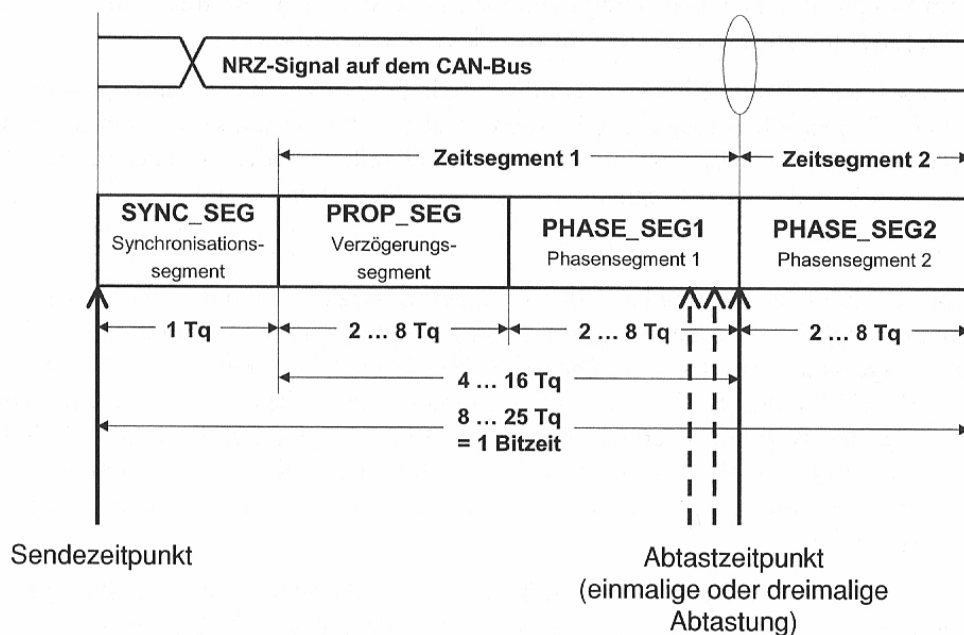


Bild 2: Aufbau einer Bitzeit

Jedes Bit, das auf dem CAN-Bus übertragen wird, kann in vier Segmente aufgeteilt werden. Die Länge der einzelnen Segmente kann im msCAN-Modul festgelegt werden und hängt zum einen von der Baudrate und zum anderen von den physikalischen Verhältnissen auf dem CAN-Bus ab. Die vier Segmente sind das Synchronisationssegment (SYNCH_SEG), das Verzögerungssegment (PROP_SEG), das Phasensegment1 (PHASE_SEG1) und das Phasensegment2 (PHASE_SEG2). Die Zeitdauer einer Bitzeit ist die Summe der vier einzelnen Segmente. Das Synchronisationssegment hat die feste Länge eines Zeitquantums T_q und stellt für jede CAN-Station den Start eines Bits dar. Für Sendestationen beginnt der neue Bitwert mit dem Synchronisationssegment. Für die Empfangsstation wird der Beginn des Bits innerhalb dieses Segments erwartet, d.h. das msCAN-Modul erwartet die erste Flanke, von der aus dann die Bitzeit gestartet wird. Auf Grund von Signalverzögerungen innerhalb des Busses ist das Synchronisationssegment zwischen dem Sender und dem Empfänger etwas verzögert. Die Verzögerungszeit hängt von der Länge zwischen dem Sender und dem Empfänger ab.

Das Verzögerungssegment kann auf die Länge 2 bis 8 Time-Quanta festgelegt werden. Es beinhaltet die physikalischen Verzögerungszeiten des CAN -Busses, des Eingangskomparators des Empfängers und des Ausgangstreibers des Senders. Es verschiebt den frühestmöglichen Zeitpunkt der Bitabtastung um den Zeitbetrag nach hinten, der gebraucht

wird, um alle Knoten im Bus zu erreichen. Das Verzögerungssegment ist auch notwendig, damit sendende Knoten beim Abarbeitungsprozess das Bit zurücklesen können. Die Phasensegmente 1 und 2 haben ebenfalls jeweils die Länge 2 bis 8 Time-Quanta. Die Verhältnisse der beiden Phasensegmente bestimmen den Abtastzeitpunkt, zu dem das CAN-Modul den Pegel des CAN-Busses einliest. Dieser Pegel stellt dann den Wert des Bits dar. Das Phasensegment 2 wird auch Zeitsegment 2 genannt.

In der msCAN-Implementierung kann das Verzögerungssegment und das Phasensegment 1 nicht getrennt eingestellt werden. Man programmiert beide zusammen als Zeitsegment 1, das dann die Länge von 4 bis 16 Time-Quanta haben kann. Der Sendezeitpunkt ist der Zeitpunkt, an der einen Sendestation einen neuen Wert auf den CAN-Bus legt. Die Empfangsstation tastet den CAN-Bus zum Abtastzeitpunkt ab. Es kann im msCAN-Modul eingestellt werden, ob einmal oder dreimal abgetastet werden soll. Bei dreimaligem Abtasten stellt der Abtastzeitpunkt die dritte Abtastung dar.

Da auf dem CAN-Bus kein Taktsignal mit übertragen wird und die einzelnen Stationen am CAN-Bus nicht hundertprozentig synchron getaktet werden, muss sich jede Empfangsstation mit jedem übertragenem Bit synchronisieren (Resynchronization). So kann sich beispielsweise die erste Flanke außerhalb des Synchronisationssegmentes befinden. Ist der Flankenwechsel vor dem Abtastzeitpunkt, spricht man von einem positiven Phasenfehler. Bei einem negativen Phasenfehler befindet sich der Flankenwechsel nach dem Abtastzeitpunkt. Der Phasenfehler ist null, wenn die Flanke innerhalb des Synchronisationssegmentes ist. Um die Flanke bei einem Phasenfehler wieder in das Synchronisationssegment zu schieben, kann das CAN-Modul die Zeitdauer des Phasensegments 1 und 2 um bis zu vier Time-Quanta verlängern und verkürzen. Die Zeitdauer der Verlängerung oder Verkürzung der beiden Phasensegmente wird Synchronisationssprungweite genannt und kann im msCAN-Modul eingestellt werden. Ist die Größe des Phasenfehlers, gemessen in Time-Quanta größer als die Synchronisationssprungweite, d. h. liegt der Flankenwechsel vor dem Abtastzeitpunkt, dann verlängert das msCAN-Modul das Phasensegment 1 um die Anzahl an SJW-Time-Quanta. Das Phasensegment 2 bleibt unverändert. Bei kleinerem Phasenfehler, bezogen auf die Synchronisationssprungweite, wird das Phasensegment 2 um die Anzahl der SJW-Time-Quanta verkürzt und das Phasensegment 1 bleibt unverändert. Damit können geringe unterschiedliche Längen einer Bitzeit von einer Sende- und Empfangsstation ausgeglichen werden. Die Synchronisationssprungweite kann im Bereich von 1 und 4 Time-Quanta eingestellt werden. Alle oben erwähnten Parameter werden in den msCAN Bustimingregistern CANBTR0 und CANBTR1 definiert.

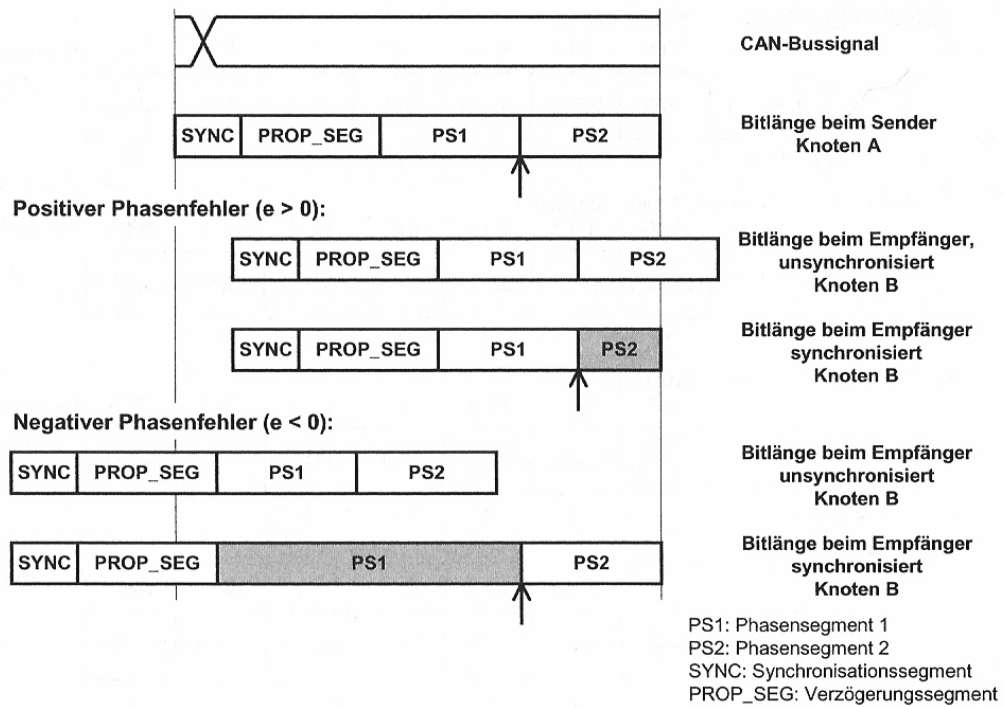


Bild 3: Beispiel für die Synchronisation bei positiven und negativen Phasenfehlern

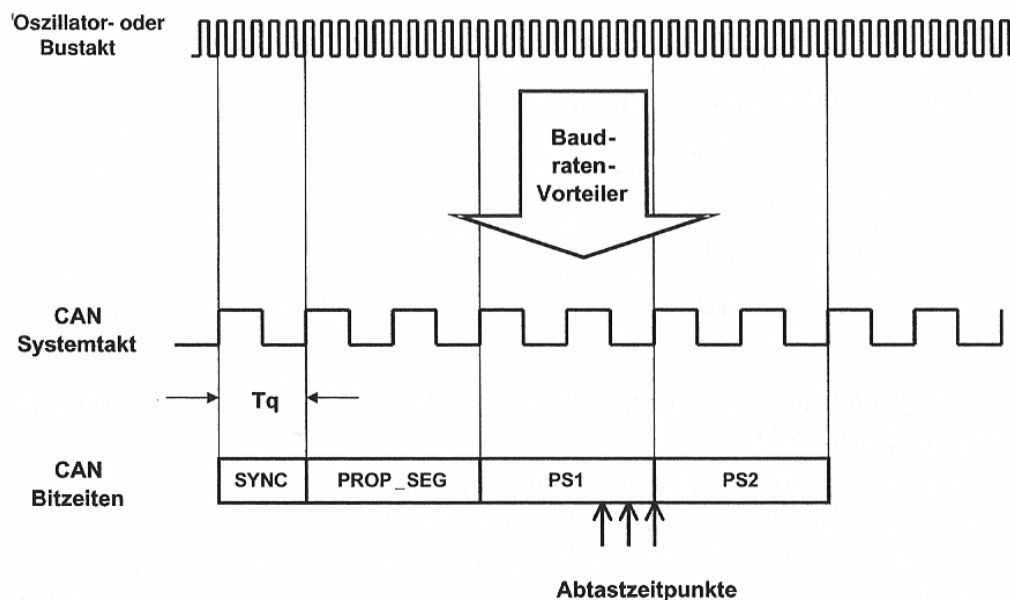


Bild 4: Zusammenhang zwischen Systemtakt, CAN- Takt, und CAN- Bitzeiten

18.7 Bestimmung der Bustiming-Parameter

Ein CAN-Bus kann auf beliebige Baudraten konfiguriert werden. Ein CAN-Controller kann bei der Konfiguration die Baudrate jedoch nicht direkt entgegennehmen, die erforderliche Einstellung muss in Form von zwei Bytes an die Bus Timing Register BTR übergeben werden. In die Berechnung der Bustiming-Parameter gehen folgende Werte ein:

- die Clock Rate des Mikrocontrollers
- die Toleranz des Oszillators, der den Controllertakt erzeugt
- die gewünschte Baudrate des CAN-Busses
- die Signalverzögerungen in der Busansteuerung
- die Buslänge
- das Abtastverhalten des CAN-Controllers (einfache oder dreifache Busabtastung)

Berechnung der Bus Timing Werte für eine Bitrate von 500 kbit/s:

Taktfrequenz: 16 MHz

Baudratenteiler: 4 $\rightarrow t_{SCL} = 1/16 \text{ MHz} * (3+1) = 250 \text{ ns}$ (CAN System Clock Period)

Synchronisationssegment: $t_{SYNCSEG} = 1 * t_{SCL}$

Zeitsegment 1: 3 $\rightarrow t_{SEG1} = t_{SCL} * (2+1) = 750 \text{ ns}$

Zeitsegment 2: 4 $\rightarrow t_{SEG2} = t_{SCL} * (3+1) = 1 \text{ us}$

Bitzeit: $t_{BIT} = t_{SYNCSEG} + t_{SEG1} + t_{SEG2} = 2 \text{ us}$

Bitrate: $f_{BIT} = 1 / t_{BIT} = 500 \text{ kbit/s}$

Daraus ergeben sich folgende Einstellungen der Register in unserem Funktionstest:

CANBTR0 = 0x03 (Synchronisationssprungweite = 1, Baudratenvorteiler = 4)

CANBTR1 = 0x32 (einfache Abtastung, Zeitsegment2 = 4, Zeitsegment1 = 3)

18.8 Funktionstest/Programmfunktionen/Registerbeschreibungen

In unserem Funktionstest für die CAN- Schnittstelle haben wir zwei HCS12 Compact über die CAN- Schnittstelle miteinander verbunden. Außerdem sind die beiden Controller jeweils über die serielle Schnittstelle mit einem PC (PC1, PC2) verbunden. Auf den PC's wurde jeweils das Hyperterminal gestartet. Geben wir nun über das Hyperterminal auf dem PC1 ein Byte ein, wird dieses über die serielle Schnittstelle auf den HCS12 Compact (1) übertragen. Dort wird das Byte über die CAN- Schnittstelle versendet und vom HCS12 Compact (2) empfangen. Auf dem HCS12 Compact (2) wird dieses Byte dann wieder über die serielle Schnittstelle zum PC2 gesendet und auf dem Hyperterminal dargestellt.

Da aber nicht immer jedem 2 HCS12 Compact zur Verfügung stehen, kam bei der Überprüfung der CAN Treiber das Programm PCANView der Firma PEAK Systems zum Einsatz. Dazu wird der PCAN-Dongle (ermöglicht den Anschluss eines CAN-Busses an die parallele Schnittstelle eines IBM kompatiblen PC's) an die parallele Schnittstelle eines PC's angeschlossen. Mit Hilfe dieses Adapters kann jeder PC in ein CAN- Netzwerk eingeschlossen werden. Der CAN- Dongle wird einfach auf den Parallelport eines PC's aufgesteckt, die Stromversorgung wird durch einen Zwischenstecker in PS/2 Ausführung am Tastaturanschluss sichergestellt. Die Hardware erfordert keine manuelle Konfiguration. Sie benötigt lediglich die Informationen über dem von ihrer parallelen Schnittstelle belegten Interrupt sowie die Portadresse (z.B. LPT1 → Adresse: 378 → Interrupt: IRQ7). Das CAN-Netz wird über den 9- poligen SUB-D Stecker angeschlossen.

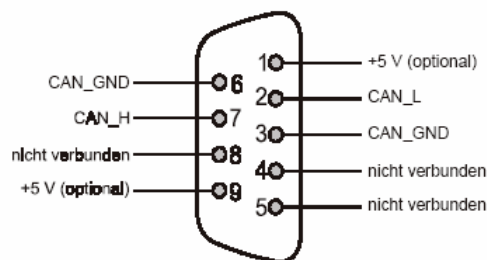


Bild 5: Anschlussbelegung 9- poligen SUB-D Stecker

Das Programm PCANView ist ein CAN-Bus Viewer. Es ermöglicht das Senden und Empfangen von CAN- Messages. Beim ersten Start werden die Portadresse sowie der Interrupt des verwendeten Parallelports angegeben (siehe Geräte-Manager von Windows). Danach wird der Typ der parallelen Schnittstelle gewählt. Ist der Modus des Parallelports im BIOS auf ECP gestellt, kann der PCAN-Dongle auch als „PEAK Dongle-CAN SJA EPP“ angemeldet werden (Bild 6).



Bild 6

Als nächstes werden die Einstellungen für den Empfangsfilter vorgenommen. In diesem Fall werden nur Nachrichten im Standard Format mit der Adresse 400h empfangen. (Bild7)

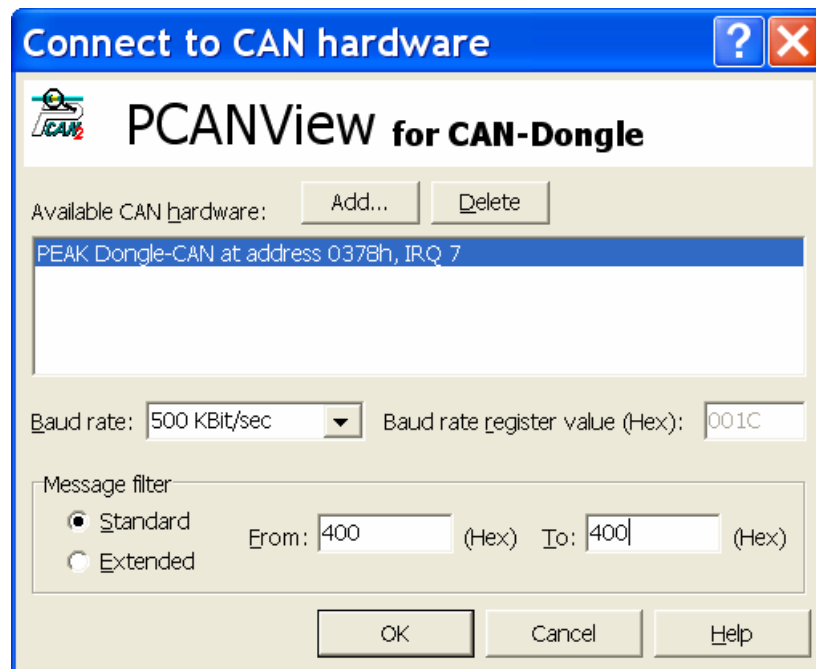


Bild 7

Die zu sendende Nachricht wird über TransmitNew eingestellt. In diesem Fall wird jede Sekunde eine Nachricht mit der ID 408h und einem Datenbyte (31 → „1“) gesendet (Bild8).

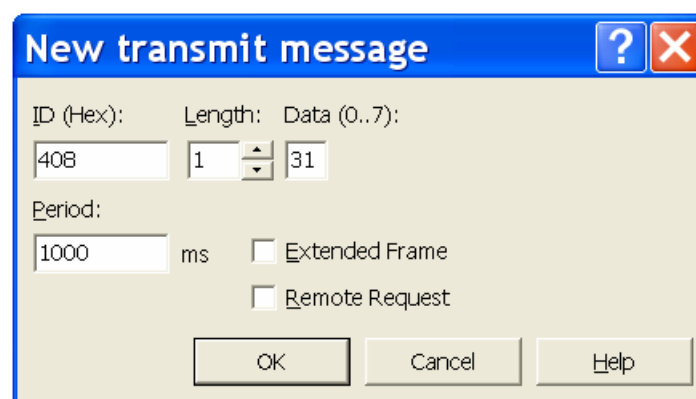


Bild 8

Hier sieht man die gesendeten und empfangenen Nachrichten.

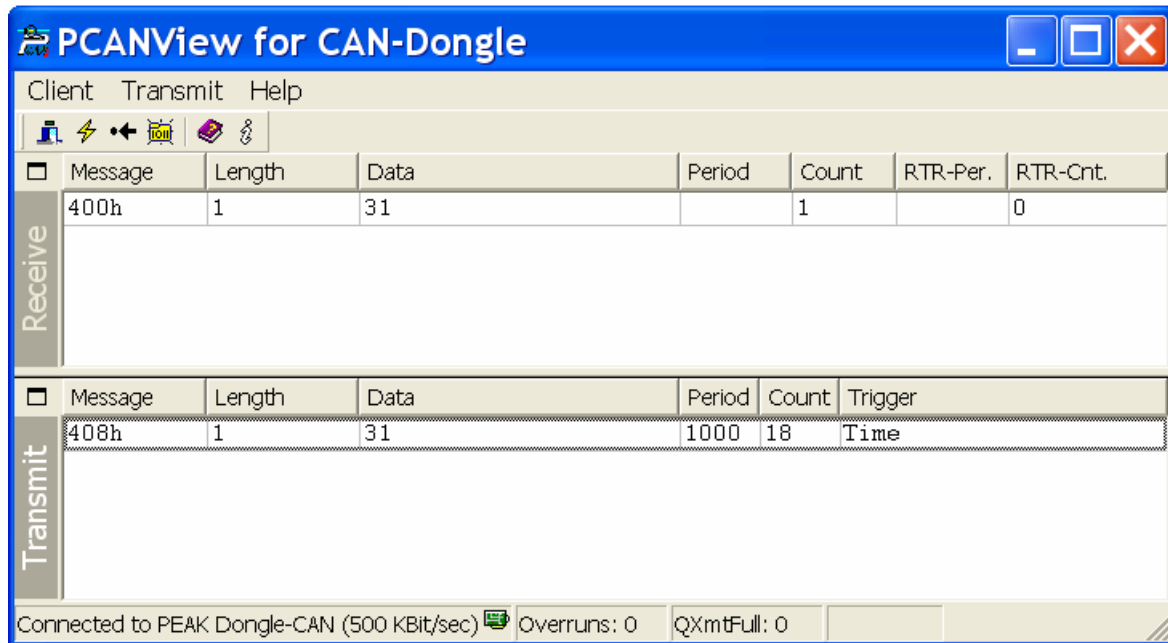


Bild 9

Quelltext: CANx_test.c

Folgende spezielle Funktionen werden im Programm verwendet:

Funktion 1: **void CANxInit(void)**

Aufgabe:	Initialisierung des jeweiligen CAN- Moduls. Hier werden die Busting Werte für eine Bitrate von 500 kbit/s und einer Taktfrequenz von 16 MHz (Die CPU wird vorher über die Funktion HCS12 Setup auf 16 MHz getaktet) gesetzt. Außerdem werden die vier 16 Bit Akzeptanzfilter (Bestimmen welche Botschaft mit welchem Identifier angenommen wird) konfiguriert. Die Maskenregister werden so gesetzt, dass das jedes Bit im Akzeptanzfilter verglichen wird.
Header:	msCANx.h
Funktionsaufruf:	CANxInit()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PortM (PM0, PM1 → CAN0), PortJ (PJ6, PJ7 → CAN4)

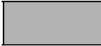
Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
CANxCTL0	Control Register0		Überwachung und Steuerung des CAN- Moduls
CAN0CTL1	Control Register1		Überwachung und Steuerung des CAN- Moduls
CAN0BTR0	Bus Timing Register 0	0x03	CAN- Baudrate 500kbps, bei 16 MHz Taktfrequenz
CAN0BTR1	Bus Timing Register 1	0x32	
CAN0IDAC	Identifier Acceptance Control Register	0x10	verwendet vier 16 Bit Akzeptanzfilter
CAN0IDMR0	Identifier Mask Register 0	0x00	keine Maskenbits bei Empfangenen Nachrichten
CAN0IDMR1	Identifier Mask Register 1	0x07	die letzten 3 Bits sind nicht benutzt
CAN0IDMR2	Identifier Mask Register 2	0x00	keine Maskenbits bei Empfangenen Nachrichten
CAN0IDMR3	Identifier Mask Register 3	0x07	die letzten 3 Bits sind nicht benutzt
CAN0IDMR4	Identifier Mask Register 4	0x00	keine Maskenbits bei Empfangenen Nachrichten
CAN0IDMR5	Identifier Mask Register 5	0x07	die letzten 3 Bits sind nicht benutzt
CAN0IDMR6	Identifier Mask Register 6	0x00	keine Maskenbits bei Empfangenen Nachrichten
CAN0IDMR7	Identifier Mask Register 7	0x07	die letzten 3 Bits sind nicht benutzt
CAN0IDAR0	Identifier Acceptance Registers 0	0x81	Setup Filter 1 (1. Filter Bank)
CAN0IDAR1	Identifier Acceptance Registers 1	0x00	
CAN0IDAR2	Identifier Acceptance Registers 2	0x81	Setup Filter 2 (1. Filter Bank)
CAN0IDAR3	Identifier Acceptance Registers 3	0x00	
CAN0IDAR4	Identifier Acceptance Registers 4	0x81	Setup Filter 3 (2. Filter Bank)
CAN0IDAR5	Identifier Acceptance Registers 5	0x00	
CAN0IDAR6	Identifier Acceptance Registers 6	0x81	Setup Filter 4 (2. Filter Bank)
CAN0IDAR7	Identifier Acceptance Register 7	0x00	
CAN0RIER	Receiver Interrupt Enable Register	0x01	Receiver Interrupt enable

Registeraufbau:

CANXCTL0 msCAN Control Register0

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	RXFRM	RXACT	CSWAI	SYNCH	TIME	WUPE	SLPRQ	INITRQ
Write:								
Reset:	0	0	0	0	0	0	0	1

 = Unimplemented

RXFRM (Empfangs-Frame Flag)

1 = eine gültige Nachricht wurde empfangen

0 = es wurde keine gültige Nachricht empfangen

RXACT (Empfänger Status)

1 = das msCAN Modul sendet eine Nachricht

0 = das msCAN modul sendet oder steht still

CSWAI (CAN geht in den Wartemodus)

1 = während des Wartemodus wird der Takt abgeschalten

0 = das Modul wird nicht bewegt während des Wartemodus

SYNCH (Synchronisationsstatus)

1 = msCAN ist mit dem CAN- Bus synchronisiert

0 = msCAN ist nicht mit dem CAN- Bus synchronisiert

TIME (Timer enable)

1 = interner msCAN Timer enable

0 = interner msCAN Timer disable

WUPE (Wake-up enable)

1 = Wake-Up enabled (msCAN kann geweckt werden)

0 = Wake-Up disabled (msCAN ignoriert Verkehr auf dem CAN Bus)

SLPRQ (Sleep Mode Anforderung)

1 = Sleep Mode Anforderung (msCAN geht in Sleep Mode wenn der Bus untätig ist)

0 = Running (msCAN funktioniert normal)

INITRQ (Initialisierungsmode Anforderung)

1 = msCAN im Initialisierungsmode

0 = normale Operation

CANxCTL1 msCAN Control Register1

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	CANE	CLKSRC	LOOPB	LISTEN	0	WUPM	SLPAK	INITAK
Write:								
Reset:	0	0	0	1	0	0	0	1

 = Unimplemented

CANE (msCAN enable)

1 = msCAN Modul enabled.

0 = msCAN Modul disabled.

CLKSRC (Quelle für den msCAN Takt)

1 = die Quelle ist der Bustakt

0 = die Quelle ist der Oszillator Takt

LOOPB (Loop Back Selbsttest Mode)

1 = Loop Back Selbsttest enabled

0 = Loop Back Selbsttest disabled

LISTEN (Horch Modus)

1 = Horch Mode ist aktiviert

0 = normale Operation

WUPM (Wake up Mode)1 = msCAN weckt die CPU bei einem Pulse von der Länge T_{WUP}

0 = msCAN weckt die CPU nach einigen Flankewechseln

SLPAK (Sleep Mode Bestätigung)

1 = msCAN befindet sich im Sleep Mode

0 = msCAN arbeitet normal

INITAK (Initialisierungsmode Bestätigung)

1 = msCAN befindet sich im Initialisierungsmode

0 = msCAN arbeitet im Normalmode

CANBTR0 msCAN Bus Timing Register 0

Address Offset: \$ _02								
	Bit 7	6	5	4	3	2	1	Bit 0
Read:	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
Write:								
Reset:	0	0	0	0	0	0	0	0

SJW1 und **SJW0**: sind zur Einstellung der Synchronisationssprungweite. Sie kann im Bereich von 1 (00) und 4 (11) Time- Quanta eingestellt werden.

BRP- Bits: damit wird der Baudratenvorteiler mit dem Teilerfaktor $P = 1$ bis 64 eingestellt. Dabei entspricht der binäre wert von BRP dem Teilerfaktor $P + 1$, d.h. $BRP = 0$ bedeutet Teilerfaktor $P = 1$ oder $BRP = 0x3f = 63$ bedeutet Teilerfaktor $P = 64$.

CANBTR1 msCAN Bus Timing Register 1

Address Offset:		\$__03						
	Bit 7	6	5	4	3	2	1	Bit 0
Read:	SAMP	TSEG22	TSEG21	TSEG20	TSEG13	TSEG12	TSEG11	TSEG10
Write:								

SAMP: Wahl zwischen einfacher (0) und dreifacher Abtastung (1)

TSEG13, TSEG12, TSEG11, TSEG10: Zeitsegment1 = Anzahl der Tq-Taktzyklen + 1

TSEG23, TSEG22, TSEG21, TSEG20: Zeitsegment2 = Anzahl der Tq-Taktzyklen + 1

CANxIDMRx msCAN Identifier Mask Registers

Read:	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset: \$__15 CANIDMR1								
Read:	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset: \$__16 CANIDMR2								
Read:	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset: \$__17 CANIDMR3								
Read:	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset: \$__1C CANIDMR4								
Read:	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset: \$__1D CANIDMR5								
Read:	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset: \$__1E CANIDMR6								
Read:	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset: \$__1F CANIDMR7								
Read:	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0
Write:								
Reset:	0	0	0	0	0	0	0	0

AM = 0 ID Bit muss mit Code übereinstimmen
 AM = 1 ID Bit ist für die Filterung nicht relevant

CANxIDARx msCAN Identifier Acceptance Registers

	Bit 7	6	5	4	3	2	1	Bit 0
Address Offset:	\$__10							CANIDAR0
Read:	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset:	\$__11							CANIDAR1
Read:	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset:	\$__12							CANIDAR2
Read:	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset:	\$__13							CANIDAR3
Read:	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 7	6	5	4	3	2	1	Bit 0
Address Offset:	\$__18							CANIDAR4
Read:	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset:	\$__19							CANIDAR5
Read:	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset:	\$__1A							CANIDAR6
Read:	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
Write:								
Reset:	0	0	0	0	0	0	0	0
Address Offset:	\$__1B							CANIDAR7
Read:	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0
Write:								
Reset:	0	0	0	0	0	0	0	0

Um 29 Bit Identifier Filtern zu können, müssen die Bits 19 (IDE) und 20 (SRR) gesetzt sein

CANIER

Address Offset:	\$__05						
	Bit 7	6	5	4	3	2	Bit 0
Read:	WUPIE	CSCIE	RSTATE1	RSTATE0	TSTATE1	TSTATE0	OVRIE
Write:							
Reset:	0	0	0	0	0	0	0

RXFIE (Wenn Empfangsregister voll wird Interrupt wird ausgelöst)

1 = wenn Empfangsregister voll wird Interrupt wird ausgelöst

0 = kein Interrupt bei vollen Empfangsregister

Funktion 2: void SendCANxMessage(UINT8 NumBytes, UINT8 *BufPntr)

Aufgabe:	Dies ist eine Funktion zum versenden einer CAN Message. Hier wird gewartet bis ein Sendepuffer verfügbar ist. Wenn ja wird dieser ausgewählt. Außerdem wird hier die Sende ID festgelegt, die mit dem Akzeptanzfilter des Empfängers übereinstimmen sollte. Danach werden die Datenbytes ins Senderegister geschrieben. Zusätzlich gibt es hier eine kleine Funktion die das Datenlängenregister beschreibt, je nach dem wie viel Bytes man übertragen möchte. Nachdem das Senderegister vollständig beschrieben wurde, wird der Sendevorgang eingeleitet.
Header:	msCANx.h
Funktionsaufruf:	SendCANxMessage(UINT8 NumBytes, UINT8 *BufPntr)
Übergabeparameter:	NumBytes → Anzahl der zu Übertragenden Bytes *BufPntr → Anfangsadresse des gepufferten Bytes
Rückgabeparameter:	keine
Reccourcen:	PortM (PM0, PM1 → CAN0), PortJ (PJ6, PJ7 → CAN4)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
CANxTFLG	Sendeflagregister		Identifizieren des verfügbaren leeren Sendepuffers
CANxTBSEL	Sendepufferauswahl		Auswahl des Sendepuffers der beschrieben werden soll
CANxTXFG	Sendepuffer		Anfangsadresse des Sendepuffers

Registeraufbau:

CANTFLG msCAN Sendeflagregister

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	0	0	0	0	0	TXE2	TXE1	TXE0
Write:								
Reset:	0	0	0	0	0	1	1	1

 = Unimplemented

TXEx (zum Identifizieren des verfügbaren leeren Sendepuffers)

1 = Sendpuffer ist verfügbar

0 = Sendpuffer ist voll

CANxTBSEL msCAN Sendepufferauswahl

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	0	0	0	0	0	TX2	TX1	TX0
Write:								
Reset:	0	0	0	0	0	0	0	0

 = Unimplemented

TXEx (erlaubt die Auswahl des Sendepuffers der beschrieben werden soll)

1 = Sendepuffer wird ausgewählt

0 = Sendepuffer nicht ausgewählt

Anm.

ist mehr als ein Bit gesetzt, wird der Puffer mit dem niederwertigeren TXx Bit ausgewählt.

CANxTXFG msCAN Sendepuffer

Addr	Register Name
\$__x0	Identifier Register 0
\$__x1	Identifier Register 1
\$__x2	Identifier Register 2
\$__x3	Identifier Register 3
\$__x4	Data Segment Register 0
\$__x5	Data Segment Register 1
\$__x6	Data Segment Register 2
\$__x7	Data Segment Register 3
\$__x8	Data Segment Register 4
\$__x9	Data Segment Register 5
\$__xA	Data Segment Register 6
\$__xB	Data Segment Register 7
\$__xC	Data Length Register
\$__xD	Transmit Buffer Priority Register ¹
\$__xE	Time Stamp Register (High Byte) ²
\$__xF	Time Stamp Register (Low Byte) ³

Interruptserviceroutine: **interrupt void CANx_Receive(void)**

Aufgabe:	Der Empfang einer CAN Nachricht wird über eine Interruptserviceroutine abgearbeitet. Diese wird gestartet, sobald eine neue Nachricht empfangen wurde. Hier werden die Bytes aus dem Empfangspuffer geholt, und nach der Zwischenspeicherung, in einer Variable, auf dem Hyperterminal dargestellt. Damit die Interruptserviceroutine gestartet werden kann, ist darauf zu achten, dass in der Interruptvektortabelle auf dem Vektor von CAN_Receive die Startadresse der Interruptserviceroutine abgelegt wird.
Header:	msCANx.h
Funktionsaufruf:	interrupt void CANx_Receive(void)
Übergabeparameter:	
Rückgabeparameter:	
Reccourcen:	PortM (PM0, PM1 → CAN0), PortJ (PJ6, PJ7 → CAN4)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
CAN0RXFG	Empfangspuffer		Empfangene Nachricht enthalten
CAN0RFLG	Empfangsflagregister		Zeigt den Zustand des Empfangspuffers an

Registeraufbau

CANxRXFG msCAN Empfangspuffer

Addr	Register Name
\$__x0	Identifier Register 0
\$__x1	Identifier Register 1
\$__x2	Identifier Register 2
\$__x3	Identifier Register 3
\$__x4	Data Segment Register 0
\$__x5	Data Segment Register 1
\$__x6	Data Segment Register 2
\$__x7	Data Segment Register 3
\$__x8	Data Segment Register 4
\$__x9	Data Segment Register 5
\$__xA	Data Segment Register 6
\$__xB	Data Segment Register 7
\$__xC	Data Length Register
\$__xD	Transmit Buffer Priority Register ¹
\$__xE	Time Stamp Register (High Byte) ²
\$__xF	Time Stamp Register (Low Byte) ³

CANRFLG msCAN Empfangsflagregister

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	WUPIF	CSCIF	RSTAT1	RSTAT0	TSTAT1	TSTAT0	OVRIFF	RXF
Write:								
Reset:	0	0	0	0	0	0	0	0

RXF (zeigt an ob eine Nachricht erhalten wurde)

1 = Empfangsfifo ist nicht leer, eine neue Nachricht kann aus dem Empfangspuffer geholt werden

0 = keine neuen Nachrichten verfügbar

19. Akustischer Signalgeber

19.1 Allgemeines

Der HCS12 Compact verfügt über einen akustischen Signalgeber SP1. Dieser wird über den Buffer IC6D getrieben und von der MCU über den Portpin PT2 angesteuert. PT2 ist mit einem der 8 Timerkanäle verbunden. Dies ermöglicht die Frequenzerzeugung mit Hilfe der Output-Compare Funktion des Timersystems.

19.2 Funktionstest/Programmfunktionen/Registerbeschreibungen

Das folgende Beispiel demonstriert, wie eine Interruptgesteuerte Output-Compare Funktion eingesetzt werden kann, um Tonfrequenzen (Polizeisirene) zu erzeugen.

Quelltext: freqout_test.c

Funktion 1: **void freqout (void)**

Aufgabe:	In dieser Funktion wird die Polizeisirene erzeugt!
Header:	freqout.h
Funktionsaufruf:	freqout()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PortP (PT2)

Aufgerufene Unterfunktionen:

Unterfunktion 1: **void delay16 (UINT 16 i)**

Aufgabe:	Integerwert wird runtergezählt
Header:	delay.h
Funktionsaufruf:	delay();
Übergabeparameter:	Integerwert 0 – 0xFFFF
Rückgabeparameter:	keine

Unterfunktion 2: **void initFreqOut(void)**

Aufgabe:	Initialisierung des Timers
Header:	freqout.h
Funktionsaufruf:	initFreqOut()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PortP (PT2)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
TSCR1	Timer System Control Register 1	0x0a	make sure timer is enabled
TSCR2	Timer System Control Register 2	0x04	prescaler = $2^{**}4 = 16$
TIOS	Timer Input Capture/Output Compare Select Register	0x02	select Output Compare function for channel 2
DDRT	PORT T Data Direction Register	0x02	PT1+PT2 output
TIE	Timer Interrupt Enable Register	0x02	enable Interrupt for channel 2
TCTL2	Timer Control Register 2	0xfD	timer disconnected from PT2 pin

Unterfunktion 3: **void setFreqOut(UINT16 period)**

Aufgabe:	In dieser Funktion wird die Frequenz erzeugt
Header:	freqout.h
Funktionsaufruf:	setFreqOut(UINT16 period)
Übergabeparameter:	Periodendauer in μs
Rückgabeparameter:	keine
Reccourcen:	PortP (PT2)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
TCTL2	Timer Control Register 2		timer disconnected or connected from PT2 pin

Interruptserviceroutine: **interrupt void isrOC2(void)**

Aufgabe:	Interruptserviceroutine wird periodisch aufgerufen
Header:	freqout.h
Funktionsaufruf:	setFreqOut(UINT16 period)
Übergabeparameter:	Periodendauer in μ s
Rückgabeparameter:	keine
Reccourcen:	PortP (PT2)

Registerwerte die für diese Funktion von Bedeutung sind:

Bezeichner	Bedeutung	Werte	Effekt
TC2	Timer Input Capture/Output Compare Register 2		timer disconnected or connected from PT2 pin
TFLG1	Main Timer Interrupt Flag 1	0x01	clear Intr flag

20. LED

20.1 Programmfunktionen

void led_on(void)

Aufgabe:	LED ein
Header:	s12_led.h
Funktionsaufruf:	led_on()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PORTE

void led_off(void)

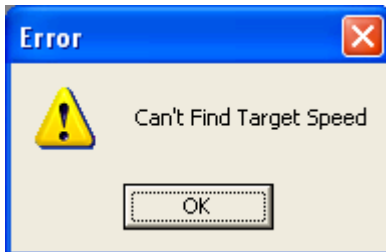
Aufgabe:	LED aus
Header:	s12_led.h
Funktionsaufruf:	led_off()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PORTE

void led_toggle (void)

Aufgabe:	LED aus
Header:	s12_led.h
Funktionsaufruf:	led_toggle()
Übergabeparameter:	keine
Rückgabeparameter:	keine
Reccourcen:	PORTE

21. Interessanter Fehler bei der Projektarbeit

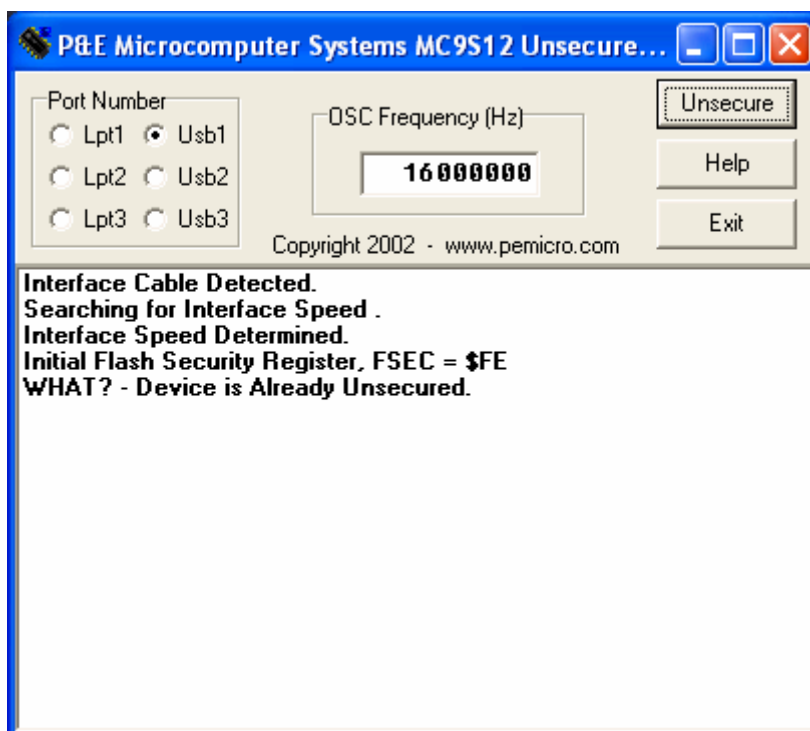
Wir hatten das Problem, dass bei ZAP BDM nach dem Übertragen an das S12Compact Modul die Fehlermeldung „Can't Find Target Speed“ angezeigt wurde.



Es war einfach nicht mehr möglich mit dem Modul in Verbindung zu kommen.

Was war passiert?

Es gab ein Problem beim Flashen, das einige Bits im Modul setzte, das so genannte **Flash Security Register**. In diesem Register kann man den Zugriff von Außen sperren, um das auslesen bzw. schreiben zu verhindern. Das geistige Eigentum des Programmierers soll so geschützt werden. Mit dem Programm „unsecure_12_install.exe“ von P&E Microcomputer Systems kann man sehr einfach das Problem beseitigen.



22. Quellcodes

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: ATD_test.c
// Datum: 2004-02-12
// Funktion: Analog Digital Wandler (10 Bit Auflösung → 5V = 1024; 0V = 0)
//           holt Wert vom Temperatursensor (1.375V = 0°C → N = 282; 3,625V = 100°C → N = 743)
//           Wert wird 1000 mal aufgenommen und dann der Durchschnitt aufs Display ausgegeben (polling)
//           Wert wird nur neu auf das Display ausgegeben wenn er sich ändert
//=====

#include <stdio.h>
#include "hcs12dp256.h"
#include "datatypes.h"
#include "Lcd.h"
#include "s12_atd.h"
//-----

void main (void)
{
    UINT8 anzeige[8], C_anz[13] = {"    Grad C"}, a=1;
    UINT16 ATD_wert=0;
    UINT32 ATD_summe, zaehl, N, wert;
    double C, Temp, merke;

    lcd_ini();                // LCD Initialisierung (0x0c als 2. Steuerbyte->Ohne Cursor)
    initATD0();               // ATD0 Initialisierung

    do                        // Endlosschleife
    {
        for(zaehl=0; zaehl<1000; zaehl++)    // Wertaufnahme und Mittelwertbildung
        {
            ATD_wert = getATD0(0);
            ATD_summe += ATD_wert;
        }
        wert = ATD_summe / 1000;
        ATD_summe = 0;

        for(C = 0, N = 282; C <= 100, N <= 743; C += 0.217, N++)
        {
            if(N == wert)
            {
                sprintf(anzeige, "%.1lf", C);    // Wert wird in Anzeigestring gewandelt
                Temp = C;
            }
        }

        /*    Temp = (wert-282)*0.217;
        sprintf(anzeige, "%.1lf", Temp);    // funktioniert auch anstatt for loop
        */
    }
}
```

```
if(a)                                     // Wert 1. mal ausgeben
{
    s_steuer(0x01);                       // 1. Zeichen, Display löschen
    for(zaehl=0; C_anz[zaehl]; zaehl++)
    {
        s_data(C_anz[zaehl]);             // Beim 1. mal "Grad C" aufs Display schreiben
    }

    s_steuer(0x02);                       // 1. Zeichen, 1. Zeile

    for(zaehl=0; anzeige[zaehl]; zaehl++)
    {
        s_data(anzeige[zaehl]);           // Temperaturanzeige
    }

    merke = Temp;                         // alten Wert merken
    a--;
}

if(merke != Temp)                        // bei verändertem Wert neue Ausgabe
{
    s_steuer(0x02);
    for(zaehl=0; anzeige[zaehl]; zaehl++)
    {
        s_data(anzeige[zaehl]);           // Temperaturanzeige
    }
}
merke = Temp;                            // alten Wert merken
}
while(1);
}
```

//=====

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: s12_atd.c
// Datum: 2004-02-12
// Funktion: Funktionen für den Analog Digital Wandler
//=====
#include "datatypes.h"
#include "hcs12dp256.h"
#include "s12_atd.h"

//-----
UINT16 atd_results[ATD_AVERAGE_COUNT][ATD_MAX_CHANNELS];
UINT16 atd_aidx;

//-----
void initATD0(void)
{
    ATD0CTL2 = BM_ADPU;           // enable ATD Modul
    ATD0CTL4 = BM_PRS2 | BM_PRS0; // 10 bit Auflösung, Clock Teiler = 12
    // 2nd sample time = 2 ATD clock
}

//-----
UINT16 getATD0(UINT8 channel)
{
    ATD0CTL3 = BM_S1C;           // wählt eine Umwandlung per Sequenz
    ATD0CTL5 = BM_DJM | (channel & 0x07); // Rechtsbündiger Vorzeichenloser Datenmode
    // Arbeitet in Einzelsequence, ein Ausgang von 8 Kanälen
    while((ATD0STAT0 & BM_SCF) == 0); // wartet bis das Sequence Complete Flag gestetzt ist
    // Vorsicht: kein Zeitschleifen Limit implementiert

    return ATD0DR0;              // liest Ergebnis Register
}

//-----
// startet Einzelkanal ATD Wandlung mit allen 8 Kanälen
// (Resultate werden später gesammelt)

void startATD0(void)
{
    ATD0CTL3 = BM_S8C;           // wählt 8 Umwandlungen per Sequenz
    ATD0CTL5 = BM_DJM | BM_MULT; // Rechtsbündiger Vorzeichenloser Datenmode
    // Arbeitet in Einzelsequence auf vielfachen Kanälen
    // (beginnend mit Kanal 0)
}

//-----
// liest Ausgangsresultate von allen 8 Kanälen
// (Wandlung wurde vorher gestartet und war erfolgreich)

BOOL readATD0(UINT16 *results)
{
    {
        UINT8 n;
        UINT16 *patd;

        if((ATD0STAT0 & BM_SCF) == 0) return FALSE; // checkt das Sequence Komplet Flag
        patd = (UINT16 *)&ATD0DR0;
        n = ATD_MAX_CHANNELS;
        while(n--) {
            // Kopiert ATD Ergebnisregister
            *results = *patd;
            results++;
            patd++;
        };
        return TRUE;
    }
}

//-----
```

```
// liest Ausgangsresultate,  
// und berechnet Gleitpunkt Durchschnitte für alle 8 Kanäle
```

```
void handleATD0(UINT16 *buf)
```

```
{  
    UINT16 *src, *dest;  
    UINT8 n, k;  
  
    dest = &atd_results[atd_aidx][0];  
    readATD0(dest); // liest Ausgangsresultate  
    startATD0(); // startet neue Sequence  
    atd_aidx++; // Incrementiert den Index  
    if(atd_aidx == ATD_AVERAGE_COUNT) atd_aidx = 0;  
  
    k = 0; // berechnet die Summe der Messungen  
    src = &atd_results[0][0]; // für jeden Kanal  
    do {  
        n = 0;  
        dest = buf;  
        do {  
            if(k == 0) *dest = *src;  
            else *dest += *src;  
            // füge die folgende Zeile zu den normalisierten Ergebnissen  
            // if(k == (ATD_AVERAGE_COUNT-1)) *dest /= ATD_AVERAGE_COUNT;  
            dest++;  
            src++;  
            n++;  
        } while(n < ATD_MAX_CHANNELS);  
        k++;  
    } while(k < ATD_AVERAGE_COUNT);  
}
```

```
//=====
```

```

//=====
// Projektarbeit "HCS12 Comact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: DA_test.c
// Function: Test DA- Wandler --> bei Eingabe eines Spannungswertes über das Hyperterminal
//           wird dieser in ein Spannungswert gewandelt (DACA oder DACB )
//=====

/* 0x0000 --> 0V
   0xffff --> 4,096 V
*/
//-----
#include <string.h>
#include <ctype.h>
#include "hcs12dp256.h"
#include "datatypes.h"
#include "DA.h"
#include "s12_crg.h"
#include "s12_sci.h"
//-----
void _HC12Setup(void)           // Startup Code
{
    COPCTL = 0x00;              // same as reset default
    initPLL();
}
//-----
void main(void)
{
    UINT8 cTaste, Kanal;
    UINT16 wert;
    DOUBLE Spannung, a;

    _HC12Setup();               // um auf 24 MHz die CPU zu takten!
    initSCI0(13);               // CPU Takt ist auf 24 MHz gestellt; 115200bps

do
{
do
{
    printf("\n\rMenue\n\rBitte geben Sie den Ausgabekanal an: ");
    printf("\n\rVOUT[A]");
    printf("\n\rVOUT[B]");
    cTaste = toupper(getchar()); // Ausgabekanal einlesen
    switch(cTaste)
    {
        case 'A': Kanal = 'A'; break;
        case 'B': Kanal = 'B'; break;
        default: printf("\n\rDie Taste \"%c\" kommt im menue nicht vor!", cTaste);
    }
    }
    while(cTaste != 'A' && cTaste != 'B');
do
{
    printf("\n\r\n\rBitte geben Sie die Spannung für diesen Kanal an!");
    printf("\n\r[ 0 ... 4.096 [V] ] :");
    scanf("%lf", &Spannung); // Spannung einlesen
    if(Spannung < 0 || Spannung > 4.096) printf("\n\rFalsche Eingabe!");
    }
    while(Spannung < 0 || Spannung > 4.096);
}

```

```
//Spannung in 16 Bit- Wert wandeln
wert = (UINT16)(Spannung / 0.0000625 + 0.5);    // N = <U/ULSB+0.5>
// for(wert = 0, a = 0; wert <= 0xffff, a <= Spannung; wert++, a = a + 0.0000625); // geht auch

//Übergabe Kanal und 16-Bit Wert an DA Wandler
putdac(Kanal, wert);
}
while(1);
}
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: DA.c
// Datum: 2004-02-12
// Funktion: Funktionen für den DA- Wandler
//=====

#include "hcs12dp256.h"
#include "datatypes.h"
#include "s12_spi.h"

//-----
void putdac(UINT8 channel, UINT16 value)
{
    SPI0CR1 = BM_SPE | BM_MSTR | BM_CPHA ; //Setup SPI-Mode, CPOL=0; CPHA=1

    // Sende Kommandos
    DDRH = 0xf0;
    PTH = 0x10; // enable D/A Wandler
    if(channel == 'A') xferSPI0(0x10); // DACA
    else xferSPI0(0x24); // DACB

    // Sende Daten
    xferSPI0(value >> 8); // Sende Bits 15..8
    xferSPI0(value); // Sende Bits 7..0
    PTH = 0x70;
    DDRH = 0x00; // disable D/A Wandler
}
//=====

//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: s12_spi.c
// Datum: 2004-02-12
// Funktion: Funktionen für die SPI- Schnittstelle
//=====

#include "datatypes.h"
#include "hcs12dp256.h"
#include "s12_spi.h"

//-----
void initSPI0(UINT8 bauddiv, UINT8 cpol, UINT8 cpha) {

    DDRS |= 0xe0; // SS,SCK,MOSI Output
    SPI0BR = bauddiv; // SPI Baudrate
    // enable SPI, Master Mode, select clock polarity/phase
    SPI0CR1 = BM_SPE | BM_MSTR | (cpol ? BM_CPOL : 0) | (cpha ? BM_CPHA : 0);
    SPI0CR2 = 0; // default
}

//-----
UINT8 xferSPI0(UINT8 abyte)
{
    SPI0DR = abyte; // starte Transfer
    while((SPI0SR & BM_SPIF) == 0); // warte bis Transfer beendet
    return(SPI0DR); // Empfangene Daten zurück
}
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: PWM_test.c
// Datum: 2004-02-12
// Funktion: Ausgabe 3 verschiedener Alarmsignale über das PWM- Modul,
//           Alarmsignal kann über das Hyperterminal geählt werden
//=====

#include <string.h>
#include "hcs12dp256.h"
#include "datatypes.h"
#include "PWM_Warnsignale.h"
#include "s12_crg.h"
#include "s12_sci.h"
//-----

void _HC12Setup(void)           // Startup Code
{
    COPCTL = 0x00;              // same as reset default
    initPLL();
}
//-----

void main(void)
{
    UINT8 cTaste;

    _HC12Setup();               // CPU auf 24 MHz
    initSCI0(13);               // 115200bps
    do
    {
        do
        {
            printf("\n\rMenue\n\rWelches Alarmsignal soll ertoenen: ");
            printf("\n\rWarnsignal[1]");
            printf("\n\rWarnsignal[2]");
            printf("\n\rWarnsignal[3]");
            cTaste = toupper(getchar());      // Ausgabekanal einlesen
            switch(cTaste)
            {
                case '1': Warnsignal1(); break;
                case '2': Warnsignal2(); break;
                case '3': Warnsignal3(); break;
                default: printf("\n\rDie Taste \"%c\" kommt im menue nicht vor!", cTaste);
            }
        }
        while(cTaste != '1' && cTaste != '2' && cTaste != '3');
    }
    while(1);
}
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: s12_PWM_Warnsignale.c
// Datum: 2004-02-12
// Funktion: PWM Initialisierung,
//           3 verschiedene Arlarmsignale über das PWM- Modul (je 3mal Piep)
//=====

#include "hcs12dp256.h"
#include "datatypes.h"
//-----

void PWM_ini (void)
{
    PWME    = 0x01;           /*PWM PP0 enable*/
    PWMPOL  = 0xff;           /*Takt startet High*/
    PWMCLK  = 0x00;           /*Clock B ist die Clock Quelle */
    PWMPRCLC = 0x00;          /*Clock B ist der Bus Takt*/
    PWMCAE  = 0x00;           /*Links ausgerichteter Ausgangsmode*/
    PWMCTL  = 0x00;           /*8-bit PWM Kanäle*/
    PWMDTY0 = 0x80;           /*Takt initialisiert auf 0x80 */
}
//-----

void PWM_disable (void)
{
    PWME    &= 0xfe;          /*PWM PP0 disable*/
}
//-----

void Warnsignal1 (void)       /*Warnungsignal: 3 kurze Töne */
{
    UINT8 w, x, y;            /*Einfache Zählvariablen*/
    PWM_ini();

    for(w=0;w<=2;w++)          /*Tönt 3 mal*/
    {
        for(x=0xFF; x>=0x01; x--) /*Schleife für die Länge des Tons*/
        {
            PWMPER0 = 0xFF;       /*PWM Periode aufs Maximum*/
            for(y=0x00; y<=0xFE; y++) /*Feste Schleife um steigenden Ton zu generieren*/
            {
                PWMDTY0 = y;       /*Läd das PWM Dienstregister*/
                Delay0(0);          /*Optionale Zeitschleife. Höherer Wert generiert längeren Ausgangston*/
            }

            for(y=0xFF; y>=0x01; y--) /*Feste Schleife um fallenden Ton zu generieren*/
            {
                PWMDTY0 = y;       /*Läd das PWM Dienstregister*/
                Delay0(0);          /*Optionale Zeitschleife. Höherer Wert generiert längeren Ausgangston*/
            }
        }
        Delay1(10000);
    }
    PWM_disable();             /*PWM PP0 disable*/
}
//-----
```

```
//-----  
void Warnsignal2 (void)          /* Warnsignal 3x Glocke */  
{  
    UINT8 w, x, y;              /*Einfache Zählvariablen*/  
    PWM_ini();  
  
    for(w=0; w<=2; w++)          /*Piept 3 mal*/  
    {  
        for(x=0xFE; x>=0x01; x--) /*Controllschleife für Signalabschwächung*/  
        {  
            PWMPER0 = x;          /*Läd das PWM Period Register*/  
  
            for(y=0; y<=x; y++)    /*Schleife für steigenden Ton*/  
            {  
                PWMDTY0 = y;      /*Läd das PWM Dienstregister*/  
                Delay0(2);         /*Optionale Zeitschleife. Höherer Wert generiert längeren Ausgangston*/  
            }  
  
            for(y=0xFF; y>=x; y--) /*Schleife für fallenden Ton*/  
            {  
                PWMDTY0 = y;      /*Läd das PWM Dienstregister*/  
                Delay0(10);       /*Optionale Zeitschleife. Höherer Wert generiert längeren Ausgangston*/  
            }  
        }  
        Delay1(3);  
        /*Zeit zwischen den Pieptönen. 0 gibt einen ununterbrochenen Ton, höherer Wert gibt grössere Zeit*/  
    }  
    PWM_disable();  
}  
//-----
```

```
//-----  
  
void Warnsignal3 (void)           /* Warnsignal */  
{  
    UINT8 w, x, y, z, k;          /*Einfache Zählvariablen*/  
    PWM_ini();  
    for(k=0; k<=2; k++)           /*Tönt 3 mal*/  
    {  
        for(w=0; w<=2; w++)  
        {  
            for(x=0xFF; x>=0x01; x--) /*Tonkontrolle für die kommende äußere Schleife*/  
            {  
                PWMPER0 = x;          /*Läd das PWM Period Register*/  
  
                for (y=0; y<=0xFE; y++) /*Ton wächst in der Schleife*/  
                PWMDTY0 = y;          /*Läd das PWM Dienstregister*/  
  
                Delay1(1);             /*Optionale/Variable delay*/  
  
                for (y=0xFF; y>=0x01; y--) /*Ton fällt in der Schleife*/  
                PWMDTY0 = y;          /*Läd das PWM Dienstregister*/  
  
                Delay1(1);             /*Optionale/Variable delay*/  
            }  
        }  
        for(z=0xFF; z>=0x01; z--) /*Ton Kontrolle fällt außerhalb der Schleife*/  
        {  
            PWMPER0 = z;              /*Läd das PWM Period Register*/  
  
            for(y=0; y<=0xFE; y++) /*Ton wächst in der Schleife*/  
            PWMDTY0 = y;            /*Läd das PWM Dienstregister*/  
  
            Delay1(1);               /*Optionale/Variable delay*/  
  
            for (y=0xFF; y>=0x01; y--) /*Ton fällt in der Schleife*/  
            PWMDTY0 = y;            /*Läd das PWM Dienstregister*/  
  
            Delay1(1);               /*Optionale/Variable delay*/  
        }  
    }  
    Delay1(10000);                 /*Wartezeit zwischen den Tönen*/  
}  
PWM_disable();  
}  
  
//-----
```

```
//-----  
  
void Delay0(int delayTime)           /*Kurz delay*/  
{  
    UINT16 x =0;                     /*äußerer Schleifenzähler */  
    UINT8 y =0;                      /*innerer Schleifenzähler */  
  
    for(x=0; x<delayTime; x++)  
    {  
        for (y=0; y<1; y++);  
    }  
}  
//-----  
  
void Delay1(int delayTime)           /*Lang delay*/  
{  
    UINT16 x =0;                     /*äußerer Schleifenzähler */  
    UINT8 y =0;                      /*innerer Schleifenzähler */  
  
    for(x=0; x<delayTime; x++)  
    {  
        for (y=0; y<100; y++);  
    }  
}  
  
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: Keypad_Lcd_poll.c
// Datum: 2004-02-12
// Funktion: Eingabe Keypad aufs Display (polling)
//=====

#include "datatypes.h"
#include "hcs12dp256.h"
#include "keypad.h"
#include "Lcd.h"
//-----
void main (void)                                // main zum testen von keypad_poll und LCD
{
    UINT16 i = 0, a = 3;
    lcd_ini();
    do
    {
        s_data(keypad_poll());
        i++;
        if(i%16==0)
        {
            if(a%2==0) s_steuer(0x01);           // 1. Zeichen 1. Zeile
            else s_steuer(0xc0);                 // 1. Zeichen 2. Zeile
            a++;
        }
    }
    while(1);
}
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: Keypad_Lcd_int.c
// Datum: 2004-02-12
// Funktion: Laufschrift läuft, während dessen Keypad Interrupt Eingabe auf 0x3000
//=====

#include "datatypes.h"
#include "hcs12dp256.h"
#include "keypad.h"
#include "Lcd.h"
#include "delay.h"
//-----

void main (void)
{
    UINT8* String ={"Projektarbeit HCS12 Compact"}, i;

    lcd_ini();                // Lcd Initialisierung
    keypad_int();              // Interrupt vorbereiten
    do
    {
        s_steuer(0x01);        // 1. Zeichen 1. Zeile + Bildschirm löschen
        for(i=0; String[i]; i++)
        {
            s_data(String[i]); // ASCII's ins Datenregister
            delay(50000);
            if(i==13) s_steuer(0xc0); // 1. Zeichen 2. Zeile
        }
    } while(1);                // Endlos
}
//=====
```

```

//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: keypad.c
// Datum: 2004-02-12
// Funktion: Keypad Funktionen
//=====

#include "hcs12dp256.h"
#include "datatypes.h"
#include "keypad.h"
#include "delay.h"

//-----
  UINT8 i;
  UINT8 keycode;

// RAFI Tatstatus (16 Tasten Hexadezimal)-----
  UINT8 Keycodes[16] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
// UINT8 Keywerte[16] = {0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf}; // Hex Werte
  UINT8 Keywerte[16] = {48,49,50,51,52,53,54,55,56,57,65,66,67,68,69,70}; // ASCII's

// Tatstatus (12 Tasten 0-9, *, #,)-----
// UINT8 Keycodes[13] = {0x01,0x02,0x03,0x05,0x06,0x07,0x09,0x0a,0x0b,0x0d,0x0e,0x0f}; // Hex Werte
// UINT8 Keywerte[13] = {'1','2','3','4','5','6','7','8','9','*','#','0'}; // ASCII's

//Polling-----

UINT8 keypad_poll (void)
{
  while((PTIP>>6 & 1)== 1); // Warte auf fallende Flanke von DA
  keycode = PTIP>>2 & 0x0f; // Inhalt vom Keypad in die Variable keycode

  for(i=0; i<16; i++) // Zuordnung ASCII Code zum Keycode
  {
    if(Keycodes[i] == keycode)
    {
      while((PTIP>>6 & 1)== 0); // Warte bis Flanke von DA wieder steigt
      return Keywerte[i]; // Rückgabewert "Keywert"
    }
  }
}

// Interrupt-----

UINT8 keypad_int (void)
{
  bMem(0x3000) = 0;
  PIEP |= 0x40; // Interrupt Enable
  PPSP &= ~0x40; // Interrupt reagiert auf fallende Flanke
  DDRP &= ~0x40; // PP6 auf Eingang
  Interruptfreigabe; // Interruptfreigabe
}

//-----

```

```
// Interruptserviceroutine-----  
  
interrupt void Port_P_ISR (void)  
{  
    UINT16 i;  
    UINT8 keycode;  
  
    // RAFI Tatstatur (16 Tasten Hexadezimal)-----  
    UINT8 Keycodes[16] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};  
    // UINT8 Keywerte[16] = {0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf}; // Hex Werte  
    UINT8 Keywerte[16] = {48,49,50,51,52,53,54,55,56,57,65,66,67,68,69,70}; // ASCII's  
  
    // Tatstatur (12 Tasten 0-9, *, #,)-----  
    // UINT8 Keycodes[13] = {0x01,0x02,0x03,0x05,0x06,0x07,0x09,0x0a,0x0b,0x0d,0x0e,0x0f}; // Hex Werte  
    // UINT8 Keywerte[13] = {'1','2','3','4','5','6','7','8','9','*','0','#'}; // ASCII's  
  
    PIFP |= 0x40; // Interruptflag zurücksetzen  
    keycode = PTIP>>2 & 0x0f; // Inhalt vom Keypad auf die Adresse 0x3000  
  
    for(i=0; i<16; i++) // Zuordnung ASCII Code zum Keycode  
    {  
        if(Keycodes[i] == keycode) *(UINT8*)(0x3000) = Keywerte[i];  
    }  
}  
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: Lcd_test.c
// Datum: 2004-02-12
// Funktion: Ausgabe Laufschrift "Projektarbeit HCS12 Compact"
//=====

#include "hcs12dp256.h"
#include "Lcd.h"
//-----

void main (void)
{
    unsigned char* String = {"Projektarbeit HCS12 Compact"}, i;

    lcd_ini();                // Lcd Initialisierung
    do
    {
        s_steuer(0x01);        // 1. Zeichen 1. Zeile + Bildschirm löschen
        for(i=0; String[i]; i++)
        {
            s_data(String[i]);  // ASCII's ins Datenregister
            delay(50000);
            if(i==13) s_steuer(0xc0); // 1. Zeichen 2. Zeile
        }
    } while(1);                // Endlos
}

//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: Lcd.c
// Datum: 2004-02-12
// Funktion: Lcd Funktionen
//
//          // E  Frei  Rs   R/W | D7  D6  D5  D4
// Beschaltung:          // PK7  x   PK5  PK4 | PK3  PK2  PK1  PK0
//=====

#include "hcs12dp256.h"
#include "datatypes.h"
#include "Lcd.h"

//-----

void lcd_ini(void)
{
    UINT16 i;

    UINT16 Steuerbytes[4] = {0x28, 0x0c, 0x01, 0x06};

    // 0x28 = Function set, 4-Bit Interface, zweizeilig, 5:7 Punktmatrix
    // 0x0E = Display ein, Cursor aus, Blinken aus
    // 0x01 = Anzeige löschen, Adresszähler 00
    // 0x06 = Entry Mode, Adresszähler autoincrement

    delay(0xffff);                // Möglichen Power On Reset abwarten
    PK_Aus;                       // PORTK = Ausgang;

    PORTK &= 0x00;
    PORTK |= 0x03;
    for(i=0; i<3; i++)            // 3 mal hintereinander eine 3 in das Steuerregister
    {
        enable();
        delay(0xffff);
    }

    s_steuer(0x02);               // Um die 4Bit Operation einzuschalten

    for(i=0; i<4; i++)            // Steuerbytes ins Steuerregister
    {
        s_steuer(Steuerbytes[i]);
    }
}

//-----

void enable (void)               // Port K kurz auf enable und wieder zurück
{
    UINT8 i;
    PORTK |= BM_7;               // enable = 1
    for(i=0; i<2; i++);         // kurz warten
    PORTK &= ~BM_7;              // enable = 0
}

//-----
```

```

void s_steuer (UINT8 s)                // Schreiben ins Steuerregister
{
    UINT8 nibble;
    PK_Aus;                            // PortK alle Ausgang
    // delay16(0xffff);                // kurz warten o.
    l_busy();                          // Busy Flag abfragen
    nibble = s>>4 & 0x0f;              // 1. Nibble herausfiltern
    PORTK &= clear;                    // PortK Clear
    PORTK |= nibble;                   // 1. Nibble auf PortK
    enable();                          // enable
    // delay16(0xffff);                // kurz warten o.
    l_busy();                          // Busy Flag abfragen
    nibble = s & 0x0f;                 // 2. Nibble herausfiltern
    PORTK &= clear;                    // PortK Clear
    PORTK |= nibble;                   // 2. Nibble auf PortK
    enable();                          // enable
    PORTK &= clear;                    // PortK Clear
}

```

```

//-----
void s_data (UINT8 s)                  // Schreiben ins Datenregister
{
    UINT8 nibble;
    PK_Aus;                            // PortK alle Ausgang
    // delay16(0xffff);                // kurz warten o.
    l_busy();                          // Busy Flag abfragen
    nibble = s>>4 & 0x0f;              // 1. Nibble herausfiltern
    PORTK &= clear;                    // PortK Clear
    PORTK |= BM_5;                     // RS = High
    PORTK |= nibble;                   // 1. Nibble auf PortK
    enable();                          // enable
    PORTK &= ~BM_5;                    // RS = Low
    // delay16(0xffff);                // kurz warten o.
    l_busy();                          // Busy Flag abfragen
    nibble = s & 0x0f;                 // 2. Nibble herausfiltern
    PORTK &= clear;                    // PortK Clear
    PORTK |= BM_5;                     // RS = High
    PORTK |= nibble;                   // 2. Nibble auf PortK
    enable();                          // enable
    PORTK &= ~BM_5;                    // RS = Low
}

```

```

//-----
void l_busy (void)                     // Busyflag lesen
{
    UINT8 a=0, b=1, i;

    DDRK = 0xf0;                       // PK0 - PK3 -->Input
    while(b)                            // Warte so lange Busy = high
    {
        PORTK = clear;
        PORTK |= BM_4;                  // R/W = 1
        PORTK |= BM_7;                  // E = 1
        a = PORTK;                      // PORTK auslesen
        PORTK &= ~BM_7;                 // E = 0
        enable();                       // E = 1 dann E = 0
        if((a >> 3 & 1) == 0) b = 0;    // wenn Busy = 0 → Abbruch
    }
    PORTK &= ~BM_4;                     // R/W = 0
    PK_Aus;                             // Port K alles Ausgang
}

```

```

//=====

```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: CAN0_test.c
// Datum: 2004-03-11
// Funktion: CAN0 test
//=====

#include "msCAN.h"
#include "hcs12dp256.h"
#include "datatypes.h"
#include "s12_crg.h"
#include "s12_sci.h"
//-----

void HCS12Setup(void);
//-----

void main(void)
{
    UINT8 Temp;

    HCS12Setup();                // CPU auf 16 MHz
    initSCI0(52);                // Setup serielle Schnittstelle 0, 19200bps

    CANInit();                  // Initialisierung des CAN- Moduls

    Interruptfreigabe;          // gebe Interrupts frei

    do
    {
        Temp = getSCI0();        // lese ein Byte von der seriellen Schnittstelle
        SendCANMessage(1, &Temp); // sende ein CAN- Paket mit dem Byte was über SCI eingegeben wurde
    }
    while(Temp !=27);            // Abbruch mit ESC, warte auf den Empfangsinterrupt,
}

//-----

void HCS12Setup(void)           // Startup Code
{
    COPCTL = 0x00;
    initPLL();                  //CPU auf 16 MHz
}

//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: msCAN0.c
// Datum: 2004-03-11
// Funktion: CAN0 Funktionen + Interruptserviceroutinen
//=====

#include "msCAN.h"
#include "hcs12dp256.h"
#include "datatypes.h"
#include "s12_sci.h"

// Interruptserviceroutine-----
// Ankommende CAN Nachrichten werden hier abgearbeitet

interrupt void CAN0_Receive(void)
{
    UINT8 Temp;

    Temp = *(CAN0RXFG+4);           // hole Empfangsbyte
    putSCI0(Temp);                  // Lokales Echo über Hyperterminal
    CAN0RFLG |= 1;                  // clear rec flag
}
// Sende Funktion -----

void SendCANMessage(UINT8 NumBytes, UINT8 *BufPntr)
{
    UINT8 NACAN, C;

    while(!CAN0TFLG);               // warte bis ein Sendepuffer verfügbar ist
    NACAN= CAN0TFLG;                // wähle den nächst verfügbaren Sendepuffer
    CAN0TBSEL= NACAN;

    *(CAN0TXFG+0)= 0x80;            // Sende ID wird festgelegt
    *(CAN0TXFG+1)= 0x00;            // 11-Bit-Identifizierformat

    if (NumBytes > 8) NumBytes = 8;
    for (C= 0; C < NumBytes; C++)   // nur so viel Bytes senden wie übergeben
        *(CAN0TXFG+4 + C)= *BufPntr++; // Bytes ins Senderegister

    *(CAN0TXFG + 0x0C)= NumBytes;   // Anzahl der Bytes ins Datenlängenregister

    NACAN = CAN0TBSEL;              // lese aktuellen Sendepuffer
    CAN0TFLG = NACAN;              // Senden wird eingeleitet
}
//-----
```

```
// CAN Initialisierung -----  
  
void CANInit(void)  
{  
    //msCAN Control Register 0  
    CAN0CTL0 |= INTRQ;           // Setze INTRQ um in den Initialisierungsmodus zu gelangen  
    while (!(CAN0CTL1 & INITAK)); // warte bis INITAK gesetzt, Initialisierungsmodus kann beginnen  
  
    //msCAN Control Register 1  
    CAN0CTL1 = CANE;             // msCAN Modul wird enabled  
  
    // Bustingregister  
    CAN0BTR0 = 0x03;  
    CAN0BTR1 = 0x32;             // CAN- Baudrate 500kbps, bei 16 MHz Taktfrequenz  
  
    //Akzeptanz Control Register  
    CAN0IDAC = 0x10;             // verwendet vier 16 Bit Akzeptanzfilter  
  
    //Akzeptanzfilterregister 0+1  
    CAN0IDAR0 = 0x81;            // Setup Filter 1 (1. Filter Bank)  
    CAN0IDAR1 = 0x00;  
  
    //Maskenregister 0+1  
    CAN0IDMR0 = 0x00;            // keine Maskenbits bei Empfangenen Nachrichten  
    CAN0IDMR1 = 0x07;            // die letzten 3 Bits sind nicht benutzt  
  
    //Akzeptanzfilterregister 2+3  
    CAN0IDAR2 = 0x81;            // Setup Filter 2 (1. Filter Bank)  
    CAN0IDAR3 = 0x00;  
  
    //Maskenregister 2+3  
    CAN0IDMR2 = 0x00;            // keine Maskenbits bei Empfangenen Nachrichten  
    CAN0IDMR3 = 0x07;            // die letzten 3 Bits sind nicht benutzt  
  
    //Akzeptanzfilterregister 4+5  
    CAN0IDAR4 = 0x81;            // Setup Filter 3 (2. Filter Bank)  
    CAN0IDAR5 = 0x00;  
  
    //Maskenregister 4+5  
    CAN0IDMR4 = 0x00;            // keine Maskenbits bei Empfangenen Nachrichten  
    CAN0IDMR5 = 0x07;            // die letzten 3 Bits sind nicht benutzt  
  
    //Akzeptanzfilterregister 6+7  
    CAN0IDAR6 = 0x81;            // Setup Filter 4 (2. Filter Bank)  
    CAN0IDAR7 = 0x00;  
  
    //Maskenregister 6+7  
    CAN0IDMR6 = 0x00;            // keine Maskenbits bei Empfangenen Nachrichten  
    CAN0IDMR7 = 0x07;            // die letzten 3 Bits sind nicht benutzt  
  
    //msCAN Control Register 0  
    CAN0CTL0 &= ~INTRQ;          // INTRQ zurücksetzen um Initialisierungsmodus zu verlassen  
    while (CAN0CTL1 & INITAK);    // warte bis INITAK zurückgesetzt, Initialisierungsmodus wird verlassen  
  
    //msCAN Control Register 1  
    CAN0CTL1 |= CANE;            // enable msCAN Modul  
  
    //Receiver Interrupt Enable Register  
    CAN0RIER = 1;                // enable receive interrupt  
}  
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: CAN4_test.c
// Datum: 2004-03-11
// Funktion: CAN4 test
//=====

#include "msCAN4.h"
#include "hcs12dp256.h"
#include "datatypes.h"
#include "s12_crg.h"
#include "s12_sci.h"

void HCS12Setup(void);
//-----

void main(void)
{
    UINT8 Temp;

    HCS12Setup();                // CPU auf 16 MHz
    initSCI(52);                // Setup serielle Schnittstelle 0, 19200bps

    CAN4Init();                 // Initialisierung des CAN- Moduls

    _asm("cli");                // gebe Interrupts frei

    do
    {
        Temp = getSCI0();        // lese ein Byte von der seriellen Schnittstelle
        SendCAN4Message(1, &Temp); // sende ein CAN- Paket mit dem Byte was über SCI eingegeben wurde
    }
    while(Temp !=27);            // Abbruch mit ESC, warte auf den Empfangsinterrupt,
}
//-----

void HCS12Setup(void)           // Startup Code
{
    COPCTL = 0x00;
    initPLL();                  // CPU auf 16 MHz
}

//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: msCAN4.c
// Datum: 2004-03-11
// Funktion: CAN4 Funktionen + Interruptserviceroutinen
//=====

#include "msCAN4.h"
#include "hcs12dp256.h"
#include "datatypes.h"
#include "s12_sci.h"

// Interruptserviceroutine-----
// Ankommende CAN Nachrichten werden hier abgearbeitet

interrupt void CAN4_Receive(void)
{
    UINT8 Temp;

    Temp = *(CAN4RXFG+4);           // hole das erste Byte
    putSCI0(Temp);                  // Lokales Echo
    CAN4RFLG |= 1;                  // clear rec flag
}

// Sende Funktion -----

void SendCAN4Message(UINT8 NumBytes, UINT8 *BufPntr)
{
    UINT8 NACAN, C;

    while(!CAN4TFLG);               // warte bis ein Sendepuffer verfügbar ist
    NACAN= CAN4TFLG;                // wähle den nächst verfügbaren Sendepuffer
    CAN4TBSEL= NACAN;

    *(CAN4TXFG+0)= 0x80;            // Sende ID wird festgelegt
    *(CAN4TXFG+1)= 0x00;            // 11-Bit-Identifizierformat

    if (NumBytes > 8) NumBytes = 8;
    for (C= 0; C < NumBytes; C++)   // nur so viel Bytes senden wie übergeben
        *(CAN4TXFG+4 + C)= *BufPntr++; // Bytes ins Senderegister

    *(CAN4TXFG + 0x0C)= NumBytes;   // Anzahl der Bytes ins Datenlängenregister

    NACAN = CAN4TBSEL;              // lese aktuellen Sendepuffer
    CAN4TFLG = NACAN;              // Senden wird eingeleitet
}

//-----
```

```
// CAN Initialisierung -----  
  
void CAN4Init(void)  
{  
    //msCAN Control Register 0  
    CAN4CTL0 |= INTRQ;           // Setze INTRQ um in den Initialisierungsmodus zu gelangen  
    while (!(CAN4CTL1 & INITAK)); // warte bis INITAK gesetzt, Initialisierungsmodus kann beginnen  
  
    //msCAN Control Register 1  
    CAN4CTL1 = CANE;             // msCAN Modul wird enabled  
  
    // Bustingregister  
    CAN4BTR0 = 0x03;  
    CAN4BTR1 = 0x32;             // CAN- Baudrate 500kbps, bei 16 MHz Taktfrequenz  
  
    //Akzeptanz Control Register  
    CAN4IDAC = 0x10;             // verwendet vier 16 Bit Akzeptanzfilter  
  
    //Akzeptanzfilterregister 0+1  
    CAN4IDAR0 = 0x81;             // Setup Filter 1 (1. Filter Bank)  
    CAN4IDAR1 = 0x00;  
  
    //Maskenregister 0+1  
    CAN4IDMR0 = 0x00;             // keine Maskenbits bei Empfangenen Nachrichten  
    CAN4IDMR1 = 0x07;             // die letzten 3 Bits sind nicht benutzt  
  
    //Akzeptanzfilterregister 2+3  
    CAN4IDAR2 = 0x81;             // Setup Filter 2 (1. Filter Bank)  
    CAN4IDAR3 = 0x00;  
  
    //Maskenregister 2+3  
    CAN4IDMR2 = 0x00;             // keine Maskenbits bei Empfangenen Nachrichten  
    CAN4IDMR3 = 0x07;             // die letzten 3 Bits sind nicht benutzt  
  
    //Akzeptanzfilterregister 4+5  
    CAN4IDAR4 = 0x81;             // Setup Filter 3 (2. Filter Bank)  
    CAN4IDAR5 = 0x00;  
  
    //Maskenregister 4+5  
    CAN4IDMR4 = 0x00;             // keine Maskenbits bei Empfangenen Nachrichten  
    CAN4IDMR5 = 0x07;             // die letzten 3 Bits sind nicht benutzt  
  
    //Akzeptanzfilterregister 6+7  
    CAN4IDAR6 = 0x81;             // Setup Filter 4 (2. Filter Bank)  
    CAN4IDAR7 = 0x00;  
  
    //Maskenregister 6+7  
    CAN4IDMR6 = 0x00;             // keine Maskenbits bei Empfangenen Nachrichten  
    CAN4IDMR7 = 0x07;             // die letzten 3 Bits sind nicht benutzt  
  
    //msCAN Control Register 0  
    CAN4CTL0 &= ~INTRQ;           // INTRQ zurücksetzen um Initialisierungsmodus zu verlassen  
    while (CAN4CTL1 & INITAK);    // warte bis INITAK zurückgesetzt, Initialisierungsmodus wird verlassen  
  
    //msCAN Control Register 1  
    CAN4CTL1 |= CANE;             // enable msCAN Modul  
  
    //Receiver Interrupt Enable Register  
    CAN4RIER |= 1;               // enable receive interrupt  
}  
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: freqout_test.c
// Datum: 2004-02-20
// Funktion: Buzzer Test (Ausgabe Polizeisirene 5mal)
//=====

#include "datatypes.h"
#include "freqout.h"

//-----

void main (void)
{
    UINT8 x=5;
    do{
        freqout();
        x--;
    }while(x);           //x mal
}
//-----
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: freqout.c
// Datum: 2004-02-12
// Funktion: Gibt eine Timer Interruptgesteuerte Polizeisirene aus
//=====

//-- Includes -----

#include "datatypes.h"
#include "hcs12dp256.h"
#include "s12_ect.h"                // contains S12_ECLK value
#include "freqout.h"
#include "delay.h"
#include "s12_crg.h"

//-- Static Vars -----

UINT16 freqout_ticks;

void initFreqOut(void)
{
    TSCR1 |= BM_TEN;                // make sure timer is enabled
    TSCR2 = 0x04;                   // prescaler = 2**4 = 16
    TIOS |= BM_2;                   // select Output Compare function for channel 2
    DDRT |= BM_2;
    TIE |= BM_2;                    // enable Interrupt for channel 2
    TCTL2 &= ~(BM_OM2 | BM_OL2);    // timer disconnected from PT2 pin
}

//-----
// period is in µs

void setFreqOut(UINT16 period)
{
    UINT16 tticks;

    tticks = period * (S12_ECLK / 2000000L);
    tticks /= TIMER_TCNT_PRE;

    if(period == 0)
    {
        TCTL2 &= ~(BM_OM2 | BM_OL2); // disconnect PT2 pin
    }
    else {
        TCTL2 |= BM_OL2;              // connect PT2 pin
    }
    freqout_ticks = tticks;
}

//-----
```

```
//-----  
  
void freqout (void)  
{  
    UINT8 i;  
  
    Interruptfreigabe;  
    initFreqOut();  
  
    for(i=0xff; i>0x6f; i--)  
    {  
        setFreqOut(i);  
        delay(1000);  
    }  
    for(i=0x6f; i<0xff;i++)  
    {  
        setFreqOut(i);  
        delay(1000);  
    }  
    initFreqOut();  
}  
  
//-----  
  
interrupt void isrOC2(void) {  
    TC2 += freqout_ticks;  
    TFLG1 = BM_2;                // clear Intr flag  
}  
  
//-----
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: Piaport.c
// Datum: 2004-02-12
// Funktion: Lauflicht Piaport
//=====

#include "hcs12dp256.h"
//-----

void main (void)
{
    unsigned char i, A, B;
    long zeit;

    DDRA = 0xff;           // Port A All Output
    DDRB = 0xff;           // Port B All Output

    do
    {
        A=0x80;           // erste LED Port A
        B=0x01;           // letzte LED Port B
        for(i=0; i<8; i++)
        {
            PORTA = A;
            PORTB = B;

            A = A>>1;       // LED shiften
            B = B<<1;       // LED shiften

            for(zeit=50000; zeit>0; zeit--); //Zeitschleife
        }
    }
    while(1);
}

//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: vectors.c
// Datum: 2004-04-03
// Funktion: Interrupt Vektortabelle, Interruptserviceroutinen
//=====

#include "keypad.h"           // Keypad Interruptserviceroutine
#include "freqout.h"          // Timer 2
#include "msCAN0.h"           // Can0
#include "msCAN4.h"           // Can4

//-----
interrupt void Dummy_ISR (void) { }

//-----

extern void _stext();          /* startup routine */

//Vektortabelle (Startadresse: ff80)-----

void (* const VectorTable[])(void) =
{
    Dummy_ISR,                // reserviert
    Dummy_ISR,                // reserviert
    Dummy_ISR,                // reserviert
    Dummy_ISR,                // reserviert
    Dummy_ISR,                // reserviert
    Dummy_ISR,                // reserviert
    Dummy_ISR,                // PWM Emergency Shutdown
    Port_P_ISR,               // Port P
    Dummy_ISR,                // CAN 4 senden
    CAN4_Receive,             // CAN 4 empfangen
    Dummy_ISR,                // CAN 4 Fehlerbehandlung
    Dummy_ISR,                // CAN 4 wakeup
    Dummy_ISR,                // CAN 3 senden
    Dummy_ISR,                // CAN 3 empfangen
    Dummy_ISR,                // CAN 3 Fehlerbehandlung
    Dummy_ISR,                // CAN 3 wakeup
    Dummy_ISR,                // CAN 2 senden
    Dummy_ISR,                // CAN 2 empfangen
    Dummy_ISR,                // CAN 2 Fehlerbehandlung
    Dummy_ISR,                // CAN 2 wakeup
    Dummy_ISR,                // CAN 1 senden
    Dummy_ISR,                // CAN 1 empfangen
    Dummy_ISR,                // CAN 1 Fehlerbehandlung
    Dummy_ISR,                // CAN 1 wakeup
    Dummy_ISR,                // CAN 0 senden
    CAN0_Receive,             // CAN 0 empfangen
    Dummy_ISR,                // CAN 0 Fehlerbehandlung
    Dummy_ISR,                // CAN 0 wakeup
    Dummy_ISR,                // Flash
    Dummy_ISR,                // EEPROM
    Dummy_ISR,                // SPI2
    Dummy_ISR,                // SPI1
    Dummy_ISR,                // IIC
    Dummy_ISR,                // BDLC wird hier nicht verwendet
    Dummy_ISR,                // SCME
    Dummy_ISR,                // CRG Lock
    Dummy_ISR,                // Pulsakkumulator B Overflow
    Dummy_ISR,                // Down Counter Underflow
    Dummy_ISR,                // Port H
    Dummy_ISR,                // Port J
}
```

```
Dummy_ISR,      // ATD1
Dummy_ISR,      // ATD2
Dummy_ISR,      // SCI1
Dummy_ISR,      // SCI2
Dummy_ISR,      // SPI0
Dummy_ISR,      // Pulsakkumulator Eingangsflanke
Dummy_ISR,      // Pulsakkumulator Überlauf
Dummy_ISR,      // Timer Überlauf
Dummy_ISR,      // Timer Kanal 7
Dummy_ISR,      // Timer Kanal 6
Dummy_ISR,      // Timer Kanal 5
Dummy_ISR,      // Timer Kanal 4
Dummy_ISR,      // Timer Kanal 3
isrOC2,         // Timer Kanal 2
Dummy_ISR,      // Timer Kanal 1
Dummy_ISR,      // Timer Kanal 0
Dummy_ISR,      // Real Time Interrupt
Dummy_ISR,      // IRQ
Dummy_ISR,      // XIRQ
Dummy_ISR,      // Software Interrupt
Dummy_ISR,      // TRAP Illegal Opcode
Dummy_ISR,      // COP Watchdog
Dummy_ISR,      // Clock Monitor
_stext          // Programmeingang, Hauptprogramm
};
```

```
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Mario Kallauke
// Datei: datatypes.h
// Datum: 2004-02-12
// Funktion: Definition der Datentypen, Bitmasken
//=====

#ifndef __DATATYPES_H
#define __DATATYPES_H

#define INT8      signed char
#define INT16     signed int
#define INT32     signed long
#define UINT8     unsigned char
#define UINT16    unsigned int
#define UINT32    unsigned long

#define VINT8     volatile signed char
#define VINT16    volatile signed int
#define VINT32    volatile signed long
#define VUINT8    volatile unsigned char
#define VUINT16   volatile unsigned int
#define VUINT32   volatile unsigned long

#define FLOAT     float
#define DOUBLE    double

#define BOOL      unsigned int
// Boolean values
#ifndef FALSE
#define FALSE 0
#define TRUE 1
#endif

// general bit masks
#define BM_7  0x80
#define BM_6  0x40
#define BM_5  0x20
#define BM_4  0x10
#define BM_3  0x08
#define BM_2  0x04
#define BM_1  0x02
#define BM_0  0x01

//Cosmic Compiler-----

//Interruptfreigabe
#define Interruptfreigabe    _asm("cli")

//Interrupt
#define interrupt            @interrupt

#endif // __DATATYPES_H =====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor(en): Mario Kallauke, Göran Liebig, Winold Doege
// Datei: hcs12dp256.h
// Datum: 2004-02-12
// Funktion: HCS12 I/O's
//=====

#ifndef __HCS12DP256_H
#define __HCS12DP256_H

/* macros, generic #define etc.
*/
// #include <hc12def.h>

/* base address of register block, change this if you relocate the register
 * block. This is for S12DP256 !
*/

#define _IO_BASE      0
#define _ADDR(off)    (unsigned char volatile *)(_IO_BASE + off)
#define _P(off)       *(unsigned char volatile *)(_IO_BASE + off)
#define _LP(off)      *(unsigned short volatile *)(_IO_BASE + off)

#define PORTA         _P(0x00)
#define PORTB         _P(0x01)
#define DDRA          _P(0x02)
#define DDRB          _P(0x03)

// #define Reserved    _P(0x04)
// #define Reserved    _P(0x05)
// #define Reserved    _P(0x06)
// #define Reserved    _P(0x07)

#define PORTE         _P(0x08)
#define DDRE          _P(0x09)
#define PEAR          _P(0x0A)
#define MODE          _P(0x0B)
#define PUCR          _P(0x0C)
#define RDRIV         _P(0x0D)
#define EBICTL        _P(0x0E)

// #define reserved    _P(0x0F)

#define INITRM        _P(0x10)
#define INITRG        _P(0x11)
#define INITEE        _P(0x12)
#define MISC          _P(0x13)
#define MTST0         _P(0x14)
#define ITCR          _P(0x15)
#define ITEST         _P(0x16)
#define MIST1         _P(0x17)

// #define Reserved    _P(0x18)
// #define Reserved    _P(0x19)

#define PARTIDH       _P(0x1A)
#define PARTIDL       _P(0x1B)
#define PARTID        _LP(0x1A)

#define MEMSIZ0       _P(0x1C)
#define MEMSIZ1       _P(0x1D)
#define INTCR         _P(0x1E)
```

```
#define HPRI0    _P(0x1F)

//#define Reserved    _P(0x20)
//#define Reserved    _P(0x21)
//#define Reserved    _P(0x22)
//#define Reserved    _P(0x23)
//#define Reserved    _P(0x24)
//#define Reserved    _P(0x25)
//#define Reserved    _P(0x26)
//#define Reserved    _P(0x27)

#define BKPCT0    _P(0x28)
#define BKPCT1    _P(0x29)
#define BKP0X     _P(0x2A)
#define BKP0H     _P(0x2B)
#define BKP0L     _P(0x2C)
#define BKP0      _LP(0x2B)

#define BKP1X     _P(0x2D)
#define BKP1H     _P(0x2E)
#define BKP1L     _P(0x2F)
#define BKP1      _LP(0x2E)

#define PPAGE     _P(0x30)

//#define Reserved    _P(0x31)

#define PORTK     _P(0x32)
#define DDRK      _P(0x33)
#define SYNRR     _P(0x34)
#define REFDV     _P(0x35)
#define CTFLG     _P(0x36)
#define CRGFLG    _P(0x37)
#define CRGINT    _P(0x38)
#define CLKSEL    _P(0x39)
#define PLLCTL    _P(0x3A)
#define RTICTL    _P(0x3B)
#define COPCTL    _P(0x3C)
#define FORBYP    _P(0x3D)
#define CTCTL     _P(0x3E)
#define ARMCOP    _P(0x3F)

#define TIOS      _P(0x40)
#define TCFORC    _P(0x41)
#define TOC7M     _P(0x42)
#define TOC7D     _P(0x43)

#define TCNT      _LP(0x44)

#define TSCR1     _P(0x46)
#define TTOV      _P(0x47)
#define TCTL1     _P(0x48)
#define TCTL2     _P(0x49)
#define TCTL3     _P(0x4A)
#define TCTL4     _P(0x4B)
#define TIE       _P(0x4C)
#define TSCR2     _P(0x4D)
#define TFLG1     _P(0x4E)
#define TFLG2     _P(0x4F)

#define TC0       _LP(0x50)
#define TC1       _LP(0x52)
#define TC2       _LP(0x54)
```

```
#define TC3    _LP(0x56)
#define TC4    _LP(0x58)
#define TC5    _LP(0x5A)
#define TC6    _LP(0x5C)
#define TC7    _LP(0x5E)

#define PACTL   _P(0x60)
#define PAFLG   _P(0x61)
#define PACN3   _P(0x62)
#define PACN2   _P(0x63)
#define PACN1   _P(0x64)
#define PACN0   _P(0x65)
#define MCCTL   _P(0x66)
#define MCFLG   _P(0x67)
#define ICPAR   _P(0x68)
#define DLYCT   _P(0x69)
#define ICOVW   _P(0x6A)
#define ICSYS   _P(0x6B)

// #define Reserved    _P(0x6C)

#define TIMTST   _P(0x6D)

// #define Reserved    _P(0x6E)
// #define Reserved    _P(0x6F)

#define PBCTL    _P(0x70)
#define PBFLG    _P(0x71)
#define PA3H     _P(0x72)
#define PA2H     _P(0x73)
#define PA1H     _P(0x74)
#define PA0H     _P(0x75)

#define MCCNT    _LP(0x76)
#define TC0H     _LP(0x78)
#define TC1H     _LP(0x7A)
#define TC2H     _LP(0x7C)
#define TC3H     _LP(0x7E)

#define ATD0CTL0 _P(0x80)
#define ATD0CTL1 _P(0x81)
#define ATD0CTL2 _P(0x82)
#define ATD0CTL3 _P(0x83)
#define ATD0CTL4 _P(0x84)
#define ATD0CTL5 _P(0x85)

#define ATD0STAT _LP(0x86)
#define ATD0STAT0 _P(0x86)
#define ATD0STAT1 _P(0x87)
#define AID0TEST0 _P(0x88)
#define AID0TEST1 _P(0x89)

// #define Reserved    _P(0x8A)
// #define Reserved    _P(0x8B)
// #define Reserved    _P(0x8C)

#define ATD0DIEN _P(0x8D)

// #define Reserved    _P(0x8E)

#define PORTAD0  _P(0x8F)

#define ATD0DR0H _P(0x90)
```

```
#define ATD0DR0L _P(0x91)
#define ATD0DR0 _LP(0x90)
```

```
#define ATD0DR1H _P(0x92)
#define ATD0DR1L _P(0x93)
#define ATD0DR1 _LP(0x92)
```

```
#define ATD0DR2H _P(0x94)
#define ATD0DR2L _P(0x95)
#define ATD0DR2 _LP(0x94)
```

```
#define ATD0DR3H _P(0x96)
#define ATD0DR3L _P(0x97)
#define ATD0DR3 _LP(0x96)
```

```
#define ATD0DR4H _P(0x98)
#define ATD0DR4L _P(0x99)
#define ATD0DR4 _LP(0x98)
```

```
#define ATD0DR5H _P(0x9A)
#define ATD0DR5L _P(0x9B)
#define ATD0DR5 _LP(0x9A)
```

```
#define ATD0DR6H _P(0x9C)
#define ATD0DR6L _P(0x9D)
#define ATD0DR6 _LP(0x9C)
```

```
#define ATD0DR7H _P(0x9E)
#define ATD0DR7L _P(0x9F)
#define ATD0DR7 _LP(0x9E)
```

```
#define PWME _P(0xA0)
#define PWMPOL _P(0xA1)
#define PWMCLK _P(0xA2)
#define PWMPRCLC _P(0xA3)
#define PWMCAE _P(0xA4)
#define PWMCTL _P(0xA5)
#define PWMTST _P(0xA6)
#define PWMPRSC _P(0xA7)
#define PWMSCLA _P(0xA8)
#define PWMSCLB _P(0xA9)
#define PWMSCNTA _P(0xAA)
#define PWMSCNTB _P(0xAB)
```

```
#define PWMCNT0 _P(0xAC)
#define PWMCNT1 _P(0xAD)
#define PWMCNT2 _P(0xAE)
#define PWMCNT3 _P(0xAF)
#define PWMCNT4 _P(0xB0)
#define PWMCNT5 _P(0xB1)
#define PWMCNT6 _P(0xB2)
#define PWMCNT7 _P(0xB3)
```

```
#define PWMPER0 _P(0xB4)
#define PWMPER1 _P(0xB5)
#define PWMPER2 _P(0xB6)
#define PWMPER3 _P(0xB7)
#define PWMPER4 _P(0xB8)
#define PWMPER5 _P(0xB9)
#define PWMPER6 _P(0xBA)
#define PWMPER7 _P(0xBB)
```

```
#define PWMDTY0 _P(0xBC)
```

```
#define PWMDTY1 _P(0xBD)
#define PWMDTY2 _P(0xBE)
#define PWMDTY3 _P(0xBF)
#define PWMDTY4 _P(0xC0)
#define PWMDTY5 _P(0xC1)
#define PWMDTY6 _P(0xC2)
#define PWMDTY7 _P(0xC3)

#define PWMSDN _P(0xC4)

// #define Reserved _P(0xC5)
// #define Reserved _P(0xC6)
// #define Reserved _P(0xC7)

#define SCI0BDH _P(0xC8)
#define SCI0BDL _P(0xC9)
#define SCI0BD _LP(0xC8)

#define SC0CR1 _P(0xCA)
#define SCI0CR1 _P(0xCA)
#define SCI0CR2 _P(0xCB)

#define SCI0SR1 _P(0xCC)
#define SC0SR1 SCI0SR1 /* compatability */
#define SC0SR2 _P(0xCD)
#define SCI0SR2 _P(0xCD)

#define SCI0DRH _P(0xCE)
#define SCI0DRL _P(0xCF)
#define SC0DRL SCI0DRL /* compatability */
#define SCI0DR _LP(0xCE)

#define SCI1BDH _P(0xD0)
#define SCI1BDL _P(0xD1)
#define SCI1BD _LP(0xD0)

#define SCI1CR1 _P(0xD2)
#define SCI1CR2 _P(0xD3)
#define SCI1SR1 _P(0xD4)
#define SCI1SR2 _P(0xD5)

#define SCI1DRH _P(0xD6)
#define SCI1DRL _P(0xD7)
#define SCI1DR _LP(0xD6)

#define SPI0CR1 _P(0xD8)
#define SPI0CR2 _P(0xD9)
#define SPI0BR _P(0xDA)
#define SPI0SR _P(0xDB)

// #define Reserved _P(0xDC)

#define SPI0DR _P(0xDD)

// #define Reserved _P(0xDE)
// #define Reserved _P(0xDF)

#define IBAD _P(0xE0)
#define IBFD _P(0xE1)
#define IBCR _P(0xE2)
#define IBSR _P(0xE3)
#define IICDR _P(0xE4)
```

```
//#define Reserved    _P(0xE5)
//#define Reserved    _P(0xE6)
//#define Reserved    _P(0xE7)

#define DLCBCR1    _P(0xE8)
#define DLCBSVR    _P(0xE9)
#define DLCBCR2    _P(0xEA)
#define DLCBDR     _P(0xEB)
#define DLCBARD    _P(0xEC)
#define DLCBRSR    _P(0xED)
#define DLCSCR     _P(0xEE)
#define DLCBSCR    _P(0xEE)
#define DLCBSTAT   _P(0xEF)

//alternate possible names for registers
#define BDLCCR1    _P(0xE8)
#define BDLCSVR    _P(0xE9)
#define BDLCCR2    _P(0xEA)
#define BDLCDR     _P(0xEB)
#define BDLCARD    _P(0xEC)
#define BDLCRSR    _P(0xED)
#define BDLCSR     _P(0xEE)
#define BDLCSTAT   _P(0xEF)

#define SPI1CR1    _P(0xF0)
#define SPI1CR2    _P(0xF1)
#define SPI1BR     _P(0xF2)
#define SPI1SR     _P(0xF3)

//#define Reserved    _P(0xF4)

#define SPI1DR     _P(0xF5)

//#define Reserved    _P(0xF6)
//#define Reserved    _P(0xF7)

#define SPI2CR1    _P(0xF8)
#define SPI2CR2    _P(0xF9)
#define SPI2BR     _P(0xFA)
#define SPI2SR     _P(0xFB)

//#define Reserved    _P(0xFC)

#define SPI2DR     _P(0xFD)

//#define Reserved    _P(0xFE)
//#define Reserved    _P(0xFF)

#define FCLKDIV    _P(0x0100)
#define FSEC       _P(0x0101)
#define C0BTR0     _P(0x0102)

//#define Reserved for Factory Test _P(0x0102)

#define FCNFG      _P(0x0103)
#define FPROT      _P(0x0104)
#define FSTAT      _P(0x0105)
#define FCMD       _P(0x0106)

//#define Reserved for Factory Test _P(0x0107)
//#define Reserved for Factory Test _P(0x0108)
//#define Reserved for Factory Test _P(0x0109)
```

```
//#define Reserved for Factory Test _P(0x010A)
//#define Reserved for Factory Test _P(0x010B)

//#define Reserved _P(0x010C)
//#define Reserved _P(0x010D)
//#define Reserved _P(0x010E)
//#define Reserved _P(0x010F)

#define ECLKDIV _P(0x0110)

//#define Reserved _P(0x0111)
//#define Reserved for Factory Test _P(0x0112)

#define ECNFG _P(0x0113)
#define EPROT _P(0x0114)
#define ESTAT _P(0x0115)
#define ECMD _P(0x0116)

//#define Reserved for Factory Test _P(0x0117)
//#define Reserved for Factory Test _P(0x0118)
//#define Reserved for Factory Test _P(0x0119)
//#define Reserved for Factory Test _P(0x011A)
//#define Reserved for Factory Test _P(0x011B)

//#define Reserved _P(0x011C)
//#define Reserved _P(0x011D)
//#define Reserved _P(0x011E)
//#define Reserved _P(0x011F)

#define ATD1CTL0 _P(0x0120)
#define ATD1CTL1 _P(0x0121)
#define ATD1CTL2 _P(0x0122)
#define ATD1CTL3 _P(0x0123)
#define ATD1CTL4 _P(0x0124)
#define ATD1CTL5 _P(0x0125)

#define ATD1STAT0 _P(0x0126)
#define ATD1STAT1 _P(0x0127)

#define ATD1TEST0 _P(0x0128)
#define ATD1TEST1 _P(0x0129)

//#define Reserved _P(0x012A)
//#define Reserved _P(0x012B)
//#define Reserved _P(0x012C)

#define ATDDIEN _P(0x012D)
#define ATD1DIEN _P(0x012D)

//#define Reserved _P(0x012E)

#define PORTAD1 _P(0x012F)

#define ATD1DR0H _P(0x0130)
#define ATD1DR0L _P(0x0131)
#define ATD1DR0 _LP(0x0130)

#define ATD1DR1H _P(0x0132)
#define ATD1DR1L _P(0x0133)
#define ATD1DR1 _LP(0x0132)

#define ATD1DR2H _P(0x0134)
#define ATD1DR2L _P(0x0135)
```

```
#define ATD1DR2  _LP(0x0134)

#define ATD1DR3H  _P(0x0136)
#define ATD1DR3L  _P(0x0137)
#define ATD1DR3  _LP(0x0136)

#define ATD1DR4H  _P(0x0138)
#define ATD1DR4L  _P(0x0139)
#define ATD1DR4  _LP(0x0138)

#define ATD1DR5H  _P(0x013A)
#define ATD1DR5L  _P(0x013B)
#define ATD1DR5  _LP(0x013A)

#define ATD1DR6H  _P(0x013C)
#define ATD1DR6L  _P(0x013D)
#define ATD1DR6  _LP(0x013C)

#define ATD1DR7H  _P(0x013E)
#define ATD1DR7L  _P(0x013F)
#define ATD1DR7  _LP(0x013E)

#define CAN0CTL0  _P(0x0140)
#define CAN0CTL1  _P(0x0141)
#define CAN0BTR0  _P(0x0142)
#define CAN0BTR1  _P(0x0143)
#define CAN0RFLG  _P(0x0144)
#define CAN0RIER  _P(0x0145)
#define CAN0TFLG  _P(0x0146)
#define CAN0TIER  _P(0x0147)
#define CAN0TARQ  _P(0x0148)
#define CAN0TAAK  _P(0x0149)

#define CAN0TBSEL _P(0x014A)
#define CAN0IDAC  _P(0x014B)

// #define Reserved _P(0x014C)
// #define Reserved _P(0x014D)

#define CAN0RXERR _P(0x014E)
#define CAN0TXERR _P(0x014F)

#define CAN0IDAR0 _P(0x0150)
#define CAN0IDAR1 _P(0x0151)
#define CAN0IDAR2 _P(0x0152)
#define CAN0IDAR3 _P(0x0153)

#define CAN0IDMR0 _P(0x0154)
#define CAN0IDMR1 _P(0x0155)
#define CAN0IDMR2 _P(0x0156)
#define CAN0IDMR3 _P(0x0157)

#define CAN0IDAR4 _P(0x0158)
#define CAN0IDAR5 _P(0x0159)
#define CAN0IDAR6 _P(0x015A)
#define CAN0IDAR7 _P(0x015B)

#define CAN0IDMR4 _P(0x015C)
#define CAN0IDMR5 _P(0x015D)
#define CAN0IDMR6 _P(0x015E)
#define CAN0IDMR7 _P(0x015F)
```

```
#define CAN0RXFG _ADDR(0x0160)
#define CAN0TXFG _ADDR(0x0170)
```

```
#define CAN1CTL0 _P(0x0180)
#define CAN1CTL1 _P(0x0181)
#define CAN1BTR0 _P(0x0182)
#define CAN1BTR1 _P(0x0183)
#define CAN1RFLG _P(0x0184)
#define CAN1RIER _P(0x0185)
#define CAN1TFLG _P(0x0186)
#define CAN1TIER _P(0x0187)
#define CAN1TARQ _P(0x0188)
#define CAN1TAAK _P(0x0189)
#define CAN1TBSEL _P(0x018A)
#define CAN1IDAC _P(0x018B)
```

```
//#define Reserved _P(0x018C)
//#define Reserved _P(0x018D)
```

```
#define CAN1RXERR _P(0x018E)
#define CAN1TXERR _P(0x018F)
```

```
#define CAN1IDAR0 _P(0x0190)
#define CAN1IDAR1 _P(0x0191)
#define CAN1IDAR2 _P(0x0192)
#define CAN1IDAR3 _P(0x0193)
#define CAN1IDMR0 _P(0x0194)
#define CAN1IDMR1 _P(0x0195)
#define CAN1IDMR2 _P(0x0196)
#define CAN1IDMR3 _P(0x0197)
#define CAN1IDAR4 _P(0x0198)
#define CAN1IDAR5 _P(0x0199)
#define CAN1IDAR6 _P(0x019A)
#define CAN1IDAR7 _P(0x019B)
#define CAN1IDMR4 _P(0x019C)
#define CAN1IDMR5 _P(0x019D)
#define CAN1IDMR6 _P(0x019E)
#define CAN1IDMR7 _P(0x019F)
```

```
#define CAN1RXFG _ADDR(0x01A0)
#define CAN1TXFG _ADDR(0x01B0)
```

```
#define CAN2CTL0 _P(0x01C0)
#define CAN2CTL1 _P(0x01C1)
#define CAN2BTR0 _P(0x01C2)
#define CAN2BTR1 _P(0x01C3)
#define CAN2RFLG _P(0x01C4)
#define CAN2RIER _P(0x01C5)
#define CAN2TFLG _P(0x01C6)
#define CAN2TIER _P(0x01C7)
#define CAN2TARQ _P(0x01C8)
#define CAN2TAAK _P(0x01C9)
#define CAN2TBSEL _P(0x01CA)
#define CAN2IDAC _P(0x01CB)
```

```
//#define Reserved _P(0x01CC)
//#define Reserved _P(0x01CD)
```

```
#define CAN2RXERR _P(0x01CE)
#define CAN2TXERR _P(0x01CF)
```

```
#define CAN2IDARO _P(0x01D0)
#define CAN2IDAR1 _P(0x01D1)
```

```
#define CAN2IDAR2 _P(0x01D2)
#define CAN2IDAR3 _P(0x01D3)

#define CAN2IDMR0 _P(0x01D4)
#define CAN2IDMR1 _P(0x01D5)
#define CAN2IDMR2 _P(0x01D6)
#define CAN2IDMR3 _P(0x01D7)

#define CAN2IDAR4 _P(0x01D8)
#define CAN2IDAR5 _P(0x01D9)
#define CAN2IDAR6 _P(0x01DA)
#define CAN2IDAR7 _P(0x01DB)

#define CAN2IDMR4 _P(0x01DC)
#define CAN2IDMR5 _P(0x01DD)
#define CAN2IDMR6 _P(0x01DE)
#define CAN2IDMR7 _P(0x01DF)

#define CAN2RXFG _ADDR(0x01E0)
#define CAN2TXFG _ADDR(0x01F0)

#define CAN3CTL0 _P(0x0200)
#define CAN3CTL1 _P(0x0201)
#define CAN3BTR0 _P(0x0202)
#define CAN3BTR1 _P(0x0203)

#define CAN3RFLG _P(0x0204)
#define CAN3RIER _P(0x0205)
#define CAN3TFLG _P(0x0206)
#define CAN3TIER _P(0x0207)
#define CAN3TARQ _P(0x0208)
#define CAN3TAAK _P(0x0209)
#define CAN3TBSEL _P(0x020A)
#define CAN3IDC _P(0x020B)

// #define Reserved _P(0x020C)
// #define Reserved _P(0x020D)

#define CAN3RXERR _P(0x020E)
#define CAN3TXERR _P(0x020F)

#define CAN3IDAR0 _P(0x0210)
#define CAN3IDAR1 _P(0x0211)
#define CAN3IDAR2 _P(0x0212)
#define CAN3IDAR3 _P(0x0213)

#define CAN3IDMR0 _P(0x0214)
#define CAN3IDMR1 _P(0x0215)
#define CAN3IDMR2 _P(0x0216)
#define CAN3IDMR3 _P(0x0217)

#define CAN3IDAR4 _P(0x0218)
#define CAN3IDAR5 _P(0x0219)
#define CAN3IDAR6 _P(0x021A)
#define CAN3IDAR7 _P(0x021B)

#define CAN3IDMR4 _P(0x021C)
#define CAN3IDMR5 _P(0x021D)
#define CAN3IDMR6 _P(0x021E)
#define CAN3IDMR7 _P(0x021F)

#define CAN3RXFG _ADDR(0x0220)
#define CAN3TXFG _ADDR(0x0230)
```

```
#define PTT    _P(0x0240)
#define PTIT   _P(0x0241)
#define DDRT   _P(0x0242)
#define RDRT   _P(0x0243)
#define PERT   _P(0x0244)
#define PPST   _P(0x0245)
```

```
//#define Reserved _P(0x0246)
//#define Reserved _P(0x0247)
```

```
#define PTS    _P(0x0248)
#define PTIS   _P(0x0249)
#define DDRS   _P(0x024A)
#define RDRS   _P(0x024B)
#define PERS   _P(0x024C)
#define PPSS   _P(0x024D)
#define WOMS   _P(0x024E)
```

```
//#define Reserved _P(0x024F)
```

```
#define PTM    _P(0x0250)
#define PTIM   _P(0x0251)
#define DDRM   _P(0x0252)
#define RDRM   _P(0x0253)
#define PERM   _P(0x0254)
#define PPSM   _P(0x0255)
#define WOMM   _P(0x0256)
```

```
//#define Reserved _P(0x0257)
```

```
#define PTP    _P(0x0258)
#define PTIP   _P(0x0259)
#define DDRP   _P(0x025A)
#define RDRP   _P(0x025B)
#define PERP   _P(0x025C)
#define PPSP   _P(0x025D)
#define PIEP   _P(0x025E)
#define PIFP   _P(0x025F)
```

```
#define PTH    _P(0x0260)
#define PTIH   _P(0x0261)
#define DDRH   _P(0x0262)
#define RDRH   _P(0x0263)
#define PERH   _P(0x0264)
#define PPSH   _P(0x0265)
#define PIEH   _P(0x0266)
#define PIFH   _P(0x0267)
```

```
#define PTJ    _P(0x0268)
#define PTIJ   _P(0x0269)
#define DDRJ   _P(0x026A)
#define RDRJ   _P(0x026B)
#define PERJ   _P(0x026C)
#define PPSJ   _P(0x026D)
#define PIEJ   _P(0x026E)
#define PIFJ   _P(0x026F)
```

```
//#define Reserved _P(0x0270 - 0x027F)
```

```
#define CAN4CTL0 _P(0x0280)
#define CAN4CTL1 _P(0x0281)
#define CAN4BTR0 _P(0x0282)
```

```
#define CAN4BTR1    _P(0x0283)
#define CAN4RFLG    _P(0x0284)
#define CAN4RIER    _P(0x0285)
#define CAN4TFLG    _P(0x0286)
#define CAN4TIER    _P(0x0287)
#define CAN4TARQ    _P(0x0288)
#define CAN4TAAK    _P(0x0289)
#define CAN4TBSEL    _P(0x028A)
#define CAN4IDAC     _P(0x028B)
```

```
//#define Reserved _P(0x028C)
//#define Reserved _P(0x028D)
```

```
#define CAN4RXERR    _P(0x028E)
#define CAN4TXERR    _P(0x028F)
```

```
#define CAN4IDAR0    _P(0x0290)
#define CAN4IDAR1    _P(0x0291)
#define CAN4IDAR2    _P(0x0292)
#define CAN4IDAR3    _P(0x0293)
```

```
#define CAN4IDMR0    _P(0x0294)
#define CAN4IDMR1    _P(0x0295)
#define CAN4IDMR2    _P(0x0296)
#define CAN4IDMR3    _P(0x0297)
```

```
#define CAN4IDAR4    _P(0x0298)
#define CAN4IDAR5    _P(0x0299)
#define CAN4IDAR6    _P(0x029A)
#define CAN4IDAR7    _P(0x029B)
```

```
#define CAN4IDMR4    _P(0x029C)
#define CAN4IDMR5    _P(0x029D)
#define CAN4IDMR6    _P(0x029E)
#define CAN4IDMR7    _P(0x029F)
```

```
#define CAN4RXFG     _ADDR(0x02A0)
#define CAN4RTFG     _ADDR(0x02B0)
```

```
//#define Reserved _P(0x02C0 - 0x02CF)
```

```
//#define Unimplemented _P(0x0300 - 0x03FF)
```

```
#endif
```

```
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Winold Doege, Mario Kallauke, Goeran Liebig
// Datei: led_und_rs232_test.c
// Datum: 2004-05-16
// Funktion: LED Menu ueber RS232 Schnittstelle
//=====

#include <stdio.h>
#include <string.h>
#include "hcs12dp256.h"
#include "s12_crg.h"
#include "datatypes.h"
#include "s12_sci.h"
#include "s12_led.h"
#include "s12_ps2.h"          //warte();
//-----
void _HC12Setup(void)
{
    COPCTL = 0x00;
    initPLL();
}
//-----
void main(void)
{
    char cTaste;
    _HC12Setup();              //um auf 16 MHz die CPU zu takten!!!!
    initSCI0(51);              //CPU Takt ist auf 16 MHz gestellt; 19600bps
                                //Baudratenteiler = Taktfrequenz : 16 : Baudrate
                                //Hyperterminal auf 19600bps einstellen!!!!

    do
    {
        printf("%c%c",0x06,0x0c); //Bildschirm loeschen
        printf("\n\rMenue\n\rBitte waehlen Sie: ");
        printf("\n\rLed ein   [1] ");
        printf("\n\rLed aus   [2] ");
        printf("\n\rLed toggle [3] ");
        cTaste = getchar();
        switch(cTaste)
        {
            case '1': led_on() ; break;
            case '2': led_off(); break;
            case '3': led_toggle(); break;
            default: printf("\n\rDie Taste \"%c\" kommt im menue nicht vor!",cTaste);
                     warte(2000000);
        }
    } while(1); //endlos schleife
}
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor(en): Winold Doege, Mario Kallauke, Göran Liebig
// Datei: s12_sci.h
// Datum: 2004-05-10
// Funktion: Initialisierung der SCI0 (RS232) und SCI1 (USB);
//           Das Holen und Senden von Bytes
//=====
#ifndef __S12_SCI_H
#define __S12_SCI_H

#define BM_LOOPS    0x80
#define BM_SCISWAI  0x40
#define BM_RSRC     0x20
#define BM_M        0x10
#define BM_WAKE     0x08
#define BM_ILT      0x04
#define BM_PE       0x02
#define BM_PT       0x01

#define BM_TIE      0x80
#define BM_TCIE     0x40
#define BM_RIE      0x20
#define BM_ILIE     0x10
#define BM_TE       0x08
#define BM_RE       0x04
#define BM_RWU      0x02
#define BM_SBK      0x01

#define BM_TDRE     0x80
#define BM_TC       0x40
#define BM_RDRF     0x20
#define BM_IDLE     0x10
#define BM_OR       0x08
#define BM_NF       0x04
#define BM_FE       0x02
#define BM_PF       0x01

#define BM_SBK13    0x04
#define BM_TXDIR    0x02
#define BM_RAF      0x01

//-----
//Prototypen

void initSCI0(UINT16 bauddiv);
UINT8 getSCI0(void);
void putSCI0(UINT8 c);

void initSCI1(UINT16 bauddiv);
UINT8 getSCI1(void);
void putSCI1(UINT8 c);
//-----
#endif /* __S12_SCI_H */

//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor(en): Winold Doege, Mario Kallauke, Göran Liebig
// Datei: s12_sci.c
// Datum: 2004-05-10
// Funktion: Initialisierung der SCI0 (RS232) und SCI1 (USB);
//           Das Holen und Senden von Bytes
//=====

#include "datatypes.h"
#include "hcs12dp256.h"
#include "s12_sci.h"

void initSCI0( UINT16 bauddiv )
{
    SCI0BD = bauddiv & 0x1fff;    // baudrate divider has 13 bits
    SCI0CR1 = 0;                 // mode = 8N1
    SCI0CR2 = BM_TE+BM_RE;       // Transmitter + Receiver enable
}

//-----
UINT8 getSCI0( void )
{
    while((SCI0SR1 & BM_RDRF) == 0) ;
    return SCI0DRL;
}

//-----

void putSCI0(UINT8 c)
{
    while((SCI0SR1 & BM_TDRE) == 0) ;
    SCI0DRL = c;
}

//-----
void initSCI1(UINT16 bauddiv)
{
    SCI1BD = bauddiv & 0x1fff;    // baudrate divider has 13 bits
    SCI1CR1 = 0;                 // mode = 8N1
    SCI1CR2 = BM_TE+BM_RE;       // Transmitter + Receiver enable
}

//-----
UINT8 getSCI1(void)
{
    while((SCI1SR1 & BM_RDRF) == 0) ;
    return SCI1DRL;
}

//-----
void putSCI1(UINT8 c)
{
    while((SCI1SR1 & BM_TDRE) == 0) ;
    SCI1DRL = c;
}
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor: Winold Doege, Mario Kallauke, Goeran Liebig
// Datei: led_und_usb_test.c
// Datum: 2004-05-16
// Funktion: LED Menu ueber USB Schnittstelle
//=====
#include <stdio.h>
#include <string.h>
#include "hcs12dp256.h"
#include "s12_crg.h"
#include "datatypes.h"
#include "s12_sci.h"
#include "s12_led.h"
#include "s12_ps2.h"          //warte();
//-----
void _HC12Setup(void) {
    COPCTL = 0x00;
    initPLL();
}
//-----
void usbprintf(const char *s) {
    int a;
    for(a=0;s[a];a++)
    {
        putSCI1(s[a]);
    }
}
//-----
char usbgetchar(void) {
    return getSCI1();
}
//-----
void main(void) {
    char cTaste, string[256] = {0};
    _HC12Setup();           //um auf 16 MHz die CPU zu takten!!!!
    initSCI1(51);           //CPU Takt ist auf 16 MHz gestellt; 19600bps
                           //Baudratenteiler = Taktfrequenz : 16 : Baudrate
                           //Hyperterminal auf 19600bps einstellen!!!!

    do
    {

        usbprintf("\n\rMenue\r\n\rBitte waehlen Sie: ");
        usbprintf("\n\rLed ein   [1] ");
        usbprintf("\n\rLed aus   [2] ");
        usbprintf("\n\rLed toggle [3] ");
        cTaste = usbgetchar();
        switch(cTaste)
        {
            case '1': led_on() ; break;
            case '2': led_off(); break;
            case '3': led_toggle(); break;
            default: usbprintf("\n\rDie Taste kommt im Menue nicht vor!");
        }
    } while(1); //endlos schleife

}
//-----
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor(en): Winold Doege, Mario Kallauke, Goeran Liebig
// Datei: ps2_test.c
// Datum: 2004-05-16
// Funktion: Die CPU ist auf 24MHz getaktet. Ein- Ausgabe ueber die PS2 Tastatur
//=====
#include "datatypes.h"
#include "hcs12dp256.h"
#include "s12_crg.h"
#include "s12_sci.h"
#include "s12_led.h"
#include "s12_ps2.h"

void _HC12Setup(void) // this func is called by the Startup Code:
{
    COPCTL = 0x00;          // same as reset default
    initPLL();
}

//-----
//Main
void main(void)
{
    _HC12Setup( );          //um auf 16 MHz die CPU zu takten!!!!
    initSCI0(51);           //CPU Takt ist auf 16 MHz gestellt; 19200bps
                           //Hyperterminal auf 19200bps einstellen!!!!
    init_ps2();             //Initialisierung der PS2 Schnittstelle
    led_on( );
    printf("%c%c",0x06,0x0c); //Bildschirm loeschen
    printf("\nHallo Welt\n");
    while(1);
}
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor(en): Winold Doege, Mario Kallauke, Göran Liebig
// Datei: s12_ps2.h
// Datum: 2004-05-16
// Funktion: PS2 auf s12Compact anschliessen
//=====

#ifndef __S12_PS2_H
#define __S12_PS2_H

#include "datatypes.h"

// 1 --> Ausgang
// 0 --> Eingang
//PJ0 --> Clock --> CLK
//PJ1 --> Data
//Ausgang
#define ACLK      0x01      //PJ0 Ausgang CLK
#define ADATA     0x02      //PJ1 Ausgang Data
#define ACLKDATA  0x03      //PJ0 und PJ1 Ausgang CLK Data

//Eingang
#define ECLK      0xFE      //PJ0 Eingang CLK
#define EDATA     0xFD      //PJ1 Eingang Data
#define ECLKDATA  0xFC      //PJ0 und PJ1 Eingang CLK Data

//Set
#define SETCLK     0x01      //Setze CLK high
#define SETDATA    0x02      //Setze DATA high
#define SETCLKDATA 0x03      //Setze CLK und Data

//Reset
#define CLRCLK     0xFE      //loesche CLK --> low
#define CLRDATA    0xFD      //loesche DATA --> low
#define CLRCLKDATA 0xFC      //loesche CLK und Data

#define CLOCKBIT   0x01      //PJ0
#define DATABIT    0x02      //PJ1
#define CLKDAT     0x03      //PJ0 und PJ1

#define CLOCKSHIFT 0          //um 0 nach rechts schieben
#define DATASHIFT  1          //um 1 nach rechts schieben
#define HIGH       1
#define LOW        0
#define ON         1
#define OFF        0

#ifndef __COSMIC_COMPILER
#define __COSMIC_COMPILER
#define ENABLE_INTERRUPTS() _asm("cli")
#define DISABLE_INTERRUPTS() _asm("sei")
#endif /*ifndef __COSMIC_COMPILER*/

//Prototypen von PS2 Schnittstelle
//-----
void init_ps2( void );      //Initialisierung der PS2 Schnittstelle
void set_CLKDAT_ON(void);   //Setzt Clock- und Datasignal auf high
void set_CLKDAT_OFF(void);  //Setzt Clock- und Datasignal auf low
void clock_on(void);        //Setzt Clocksignal auf low
```

```
void clock_off(void);      //Setzt Clocksignal auf high
void data_on(void);       //Setzt Datasignal auf low
void data_off(void);      //Setzt Datasignal auf high
void set_clock(UINT8 Bit); //Setzt je nach Bit Clocksignal
void set_data(UINT8 Bit);  //Setzt je nach Bit Datasignal
int read_clock(void);      //Lese Clocksignal
int read_data(void);       //Lese Datasignal
interrupt void kb_isr(void); //Interrupt Routine
void warte(volatile unsigned long wert); //Verzögerung um dem Wert x
unsigned char kb_out(void); //Lesen eines Bytes aus der Tastatur
void kb_in(UINT8 byte);     //schreiben eines Bytes in die Tastatur
unsigned char scancode_ascii(unsigned char code); //Scancode Zeichen -> ASCII
```

```
//-----
#endif /* __S12_PS2_H */
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor(en): Winold Doege, Mario Kallauke, Göran Liebig
// Datei: s12_ps2.c
// Datum: 2004-05-16
// Funktion: PS2 an S12Compact anschliessen
//=====
#include <stdio.h>
#include "s12_ps2.h"
#include "hcs12dp256.h"
#include "datatypes.h"
//-----
void init_ps2( void ) {                                //Initialisierung der PS2 Schnittstelle
    DISABLE_INTERRUPTS();                               //PJ1 auf Eingang
    DDRJ &= EDATA;                                     //Datasignal loest Interrupt aus
    PIEJ |= SETDATA;                                   //Interruptfreigabe
    ENABLE_INTERRUPTS();
}
//-----
void set_CLKDAT_ON(void) {                             //Setzt Clock- und Datasignal auf high
    DDRJ |= ACLKDATA;                                  //PJ0 und PJ1 Ausgang CLK Data
    PTJ |= SETCLKDATA;                                //PJ0 und PJ1 high
}
//-----
void set_CLKDAT_OFF(void) {                             //Setzt Clock- und Datasignal auf low
    DDRJ |= ACLKDATA;                                  //PJ0 und PJ1 Ausgang CLK Data
    PTJ &= CLRCLKDATA;                                //PJ0 und PJ1 low
}
//-----
void clock_on(void) {                                   //Setzt Clocksignal auf high
    DDRJ |= ACLK;                                       //PJ0 Ausgang CLK
    PTJ |= SETCLK;                                     //Setzt CLOCK auf high
}
//-----
void clock_off(void) {                                  //Setzt Clocksignal auf low
    DDRJ |= ACLK;                                       //PJ0 Ausgang CLK
    PTJ &= CLRCLK;                                     //loesche CLK --> low
}
//-----
void data_on(void) {                                    //Setzt Datasignal auf high
    DDRJ |= ADATA;                                     //PJ1 Ausgang Data
    PTJ |= SETDATA;                                    //Setze DATA high
}
//-----
void data_off(void) {                                   //PJ1 Ausgang Data
    DDRJ |= ADATA;                                     //loesche DATA --> low
    PTJ &= CLRDATA;
}
//-----
void set_clock(UINT8 Bit) {                             //Setzt je nach Bit Clocksignal
    if (Bit)
    {
        DDRJ |= ACLK;                                  //PJ0 Ausgang CLK
        PTJ |= SETCLK;                                //Setzt CLOCK auf high
    }
    else
    {
        DDRJ |= ACLK;                                  //PJ0 Ausgang CLK
        PTJ &= CLRCLK;                                //loesche CLK --> low
    }
}
}
```

```
//-----
void set_data(UINT8 Bit) {                               //Setzt je nach Bit Datasignal
  if (Bit)
  {
    DDRJ |= ADATA;                                       //PJ1 Ausgang Data
    PTJ |= SETDATA;                                     //Setze DATA high
  }
  else
  {
    DDRJ |= ADATA;                                       //PJ1 Ausgang Data
    PTJ &= CLRDATA;                                     //loesche DATA --> low
  }
}
//-----
int read_clock(void)                                     //Lese Clocksignal
{
  return (PTIJ & CLOCKBIT);
}
//-----
int read_data(void)                                     //Lese Datasignal
{
  return (((PTIJ & DATABIT)) >> DATASHIFT );
}
//-----
unsigned char kb_out(void)                             //Tastatur --> Mikroprozessor
{
  unsigned char byte = 0, stopbit,paritybit;
  INT8 a=0, parity, ParityBit = 0;

  while( (read_clock() == LOW) && (read_data() == LOW));
  while( read_clock() == HIGH);

  for( a=0,byte=0,parity=0;a<=9;a++) // Bits einlesen
  {
    while(read_clock() == LOW );
    while(read_clock() == HIGH );
    byte = byte | (read_data() << a);
    if( a <= 7)
    {
      if( read_data() == 1 )
      {
        parity++;
      }
    }
    if( a == 8) paritybit = read_data();
    if( a == 9) stopbit = read_data();
  }

  if( stopbit == 0)
  {
    if( parity%2 == 0)
    {
      ParityBit = 1;                                     // 0 ist Vorgabe siehe oben
    }

    if(ParityBit == paritybit)
    {
      byte = 0x00;
    }
  }
  return byte;
}
```

```
//-----
void kb_in(UINT8 byte) //Rechner --> Tastatur
{
    UINT8 Bit[11] = {0,0,0,0,0,0,0,0,0,1,0};
    volatile unsigned n = 0;
    int a=0, parity=0, ParityBit = 0, b=0;

    for(a=0 ; a<=7 ; a++)
    {
        Bit[a] = ((byte >> a) & 0x01);
        if(Bit[a] == 1) parity++;
    }
    if( parity%2 == 0) Bit[8] = 1;    // 0 ist Vorgabe siehe oben
    data_off();
    warte(200);                      //warte ca. xxx uS
    while(read_clock()==HIGH);
    for(a=0; a<=9; a++)
    {
        do {
            set_data( Bit[a]);
        } while( read_clock() == LOW );
        while( read_clock() == HIGH );
    }
    ENABLE_INTERRUPTS();
}
//-----
// Geforderte Zeit durch 1.5us = wert
void warte(volatile unsigned long wert)
{
    while(--wert);
}
//-----
interrupt void kb_isr(void)
{
    unsigned cvar;
    DISABLE_INTERRUPTS();                //Alle Interrupts sperren
    cvar = kb_out();                      //hole Zeichen von der Schnittstelle
    if(cvar == 0xf0)
    {
        warte(12000);                    //um das Zeite Zeichen zu schlucken
                                         //f0 xx
    }
    else
    {
        printf("%c",scancode_ascii(cvar)); //Ausgabe auf dem Bildschirm
        //printf("\nAls Hex %02x als Zeichen %c",cvar, scancode_ascii(cvar));
    }
    ENABLE_INTERRUPTS();                  //Interrupts wieder freigeben
}
//-----
unsigned char scancode_ascii(unsigned char code)
{
    unsigned char symbol;
    switch(code)
    {
        case 0x29 :    symbol = ' '; break;
        case 0x5a :    symbol = '\n'; break;
        case 0x0e :    symbol = '^'; break;
        case 0x16 :    symbol = '1'; break;
        case 0x1e :    symbol = '2'; break;
        case 0x26 :    symbol = '3'; break;
        case 0x25 :    symbol = '4'; break;
    }
}
```

```
case 0x2e :    symbol = '5'; break;
case 0x36 :    symbol = '6'; break;
case 0x3d :    symbol = '7'; break;
case 0x3e :    symbol = '8'; break;
case 0x46 :    symbol = '9'; break;
case 0x45 :    symbol = '0'; break;
case 0x4e :    symbol = 'ß'; break;
case 0x55 :    symbol = '\"; break;
case 0x66 :    symbol = 8 ; break;
case 0x15 :    symbol = 'Q'; break;
case 0x1d :    symbol = 'W'; break;
case 0x24 :    symbol = 'E'; break;
case 0x2d :    symbol = 'R'; break;
case 0x2c :    symbol = 'T'; break;
case 0x35 :    symbol = 'Z'; break;
case 0x3c :    symbol = 'U'; break;
case 0x43 :    symbol = 'I'; break;
case 0x44 :    symbol = 'O'; break;
case 0x4d :    symbol = 'P'; break;
case 0x54 :    symbol = 'Ü'; break;
case 0x5b :    symbol = '+'; break;
case 0x1c :    symbol = 'A'; break;
case 0x1b :    symbol = 'S'; break;
case 0x23 :    symbol = 'D'; break;
case 0x2b :    symbol = 'F'; break;
case 0x34 :    symbol = 'G'; break;
case 0x33 :    symbol = 'H'; break;
case 0x3b :    symbol = 'J'; break;
case 0x42 :    symbol = 'K'; break;
case 0x4b :    symbol = 'L'; break;
case 0x4c :    symbol = 'Ö'; break;
case 0x52 :    symbol = 'Ä'; break;
case 0x5d :    symbol = '#'; break;
case 0x13 :    symbol = '<'; break;
case 0x1a :    symbol = 'Y'; break;
case 0x22 :    symbol = 'X'; break;
case 0x21 :    symbol = 'C'; break;
case 0x2a :    symbol = 'V'; break;
case 0x32 :    symbol = 'B'; break;
case 0x31 :    symbol = 'N'; break;
case 0x3a :    symbol = 'M'; break;
case 0x41 :    symbol = ','; break;
case 0x49 :    symbol = '.'; break;
case 0x4a :    symbol = '-'; break;
case 0x6c :    symbol = '7'; break;
case 0x6b :    symbol = '4'; break;
case 0x69 :    symbol = '1'; break;
case 0x77 :    symbol = '/'; break;
case 0x75 :    symbol = '8'; break;
case 0x73 :    symbol = '5'; break;
case 0x72 :    symbol = '2'; break;
case 0x70 :    symbol = '0'; break;
case 0x7e :    symbol = '*'; break;
case 0x7d :    symbol = '9'; break;
case 0x74 :    symbol = '6'; break;
case 0x7a :    symbol = '3'; break;
case 0x84 :    symbol = '-'; break;
case 0x61 :    symbol = '|'; break;
case 0x79 :    symbol = '+'; break;
case 0x7b :    symbol = '-'; break;
case 0x7c :    symbol = '*'; break;
case 0x71 :    symbol = ','; break;
default: break;
```

```
    }
    return symbol;
}
/*
void delay(void)
{
    volatile unsigned n = 0;
    while(--n) ;
}
*/
```

```
//=====
```

```
//=====
// Projektarbeit "HCS12 Compact" Staatliche Technikerschule Berlin
// Autor(en): Winold Doege, Mario Kallauke, Goeran Liebig
// Datei: rechteckgenerator.c
// Datum: 2004-05-18
// Funktion: 1kHz sym. Rechtecksignal am Ausgang 1 bis 4
// Die CPU ist auf 8MHz getaktet
// Im Register TSCR2 ist der Teiler 32 eingestellt.
//=====
```

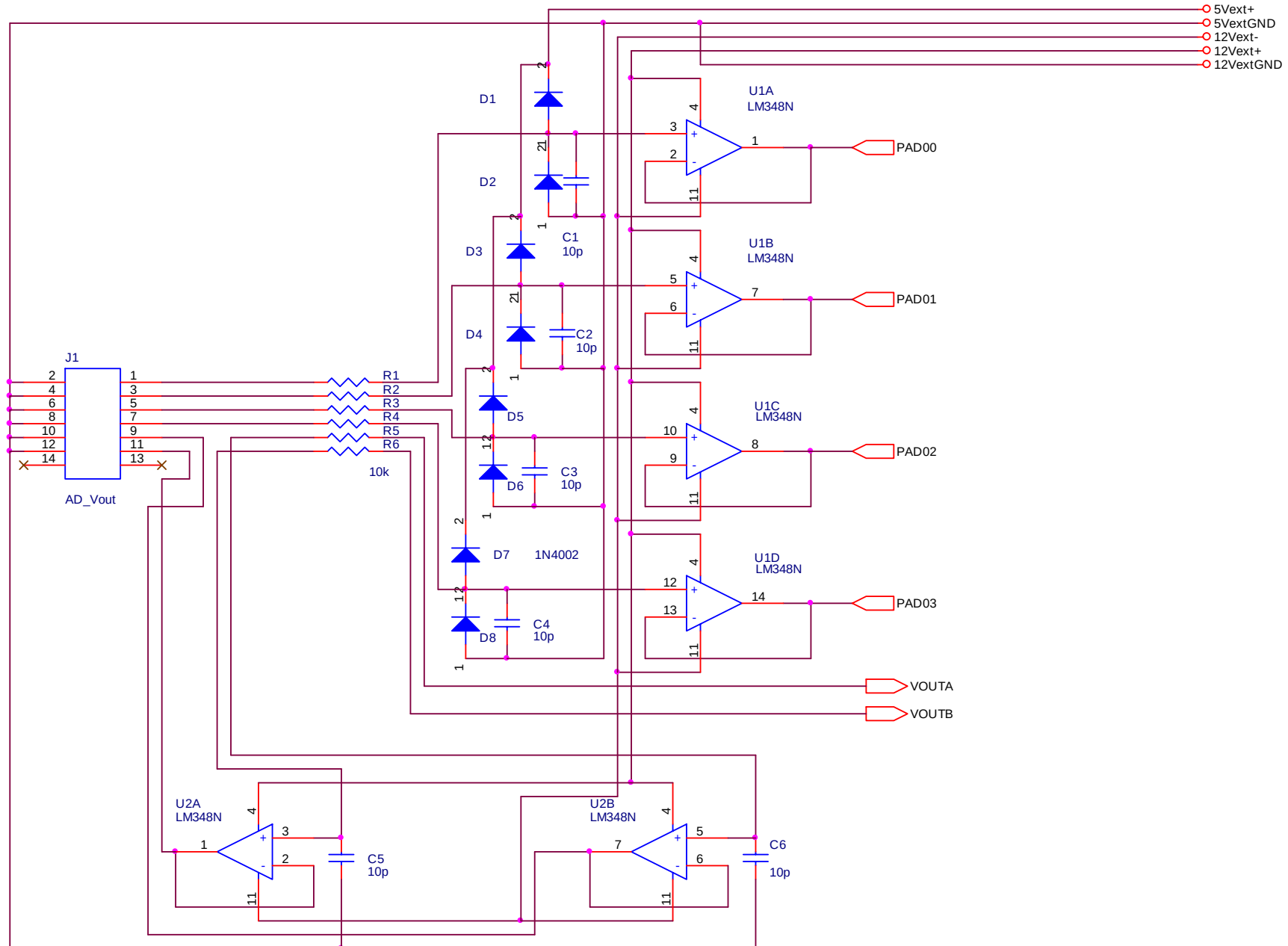
```
#include "datatypes.h"
#include "hcs12dp256.h"
```

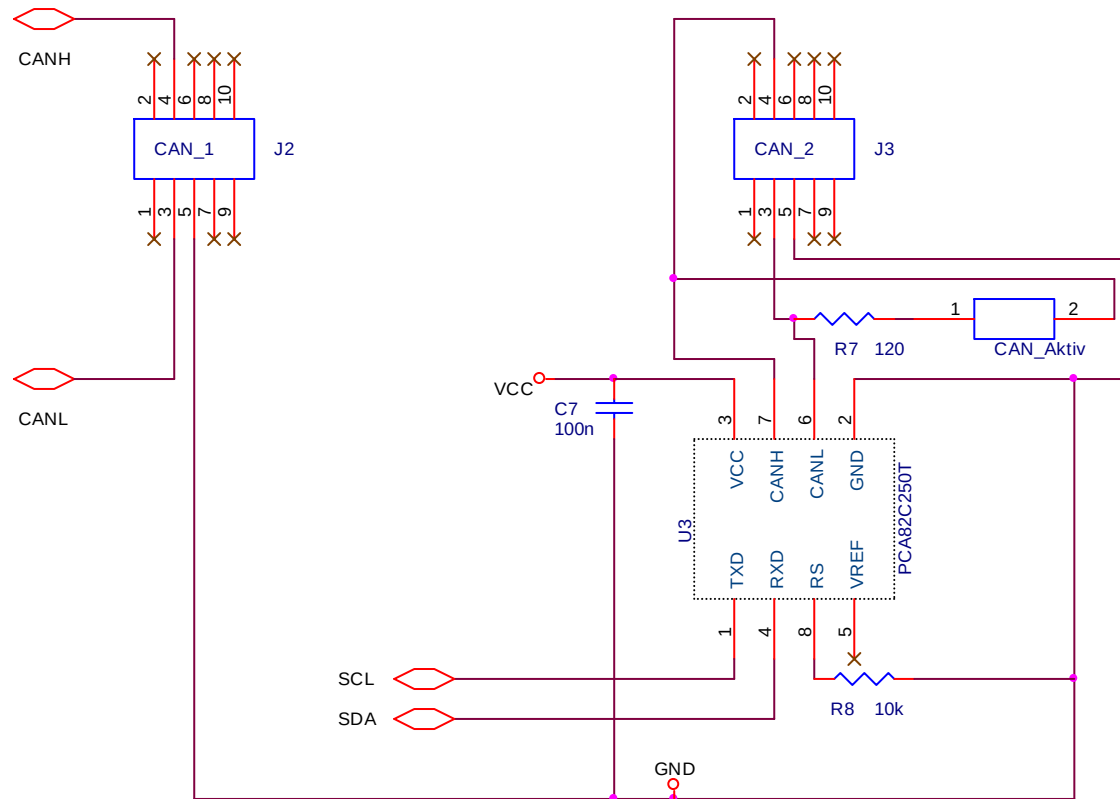
```
//-----
void main(void)
{
    TIOS = 0xff;
    TSCR1 = 0x80;
    TSCR2 = 0x0d;

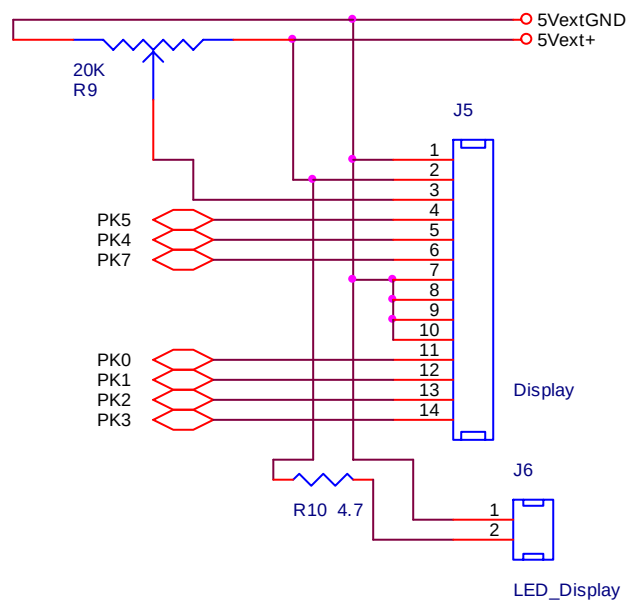
    RDRT = 0x0F;
    TFLG1 = 0xff;
    TOC7M = 0x0F;
    TOC7D = 0x0F;
    TCTL2 = 0xAA;
    TC0=127;
    TC1=127;
    //TC2=127; //MCU Hupe
    TC3=127;
    TC7=255;

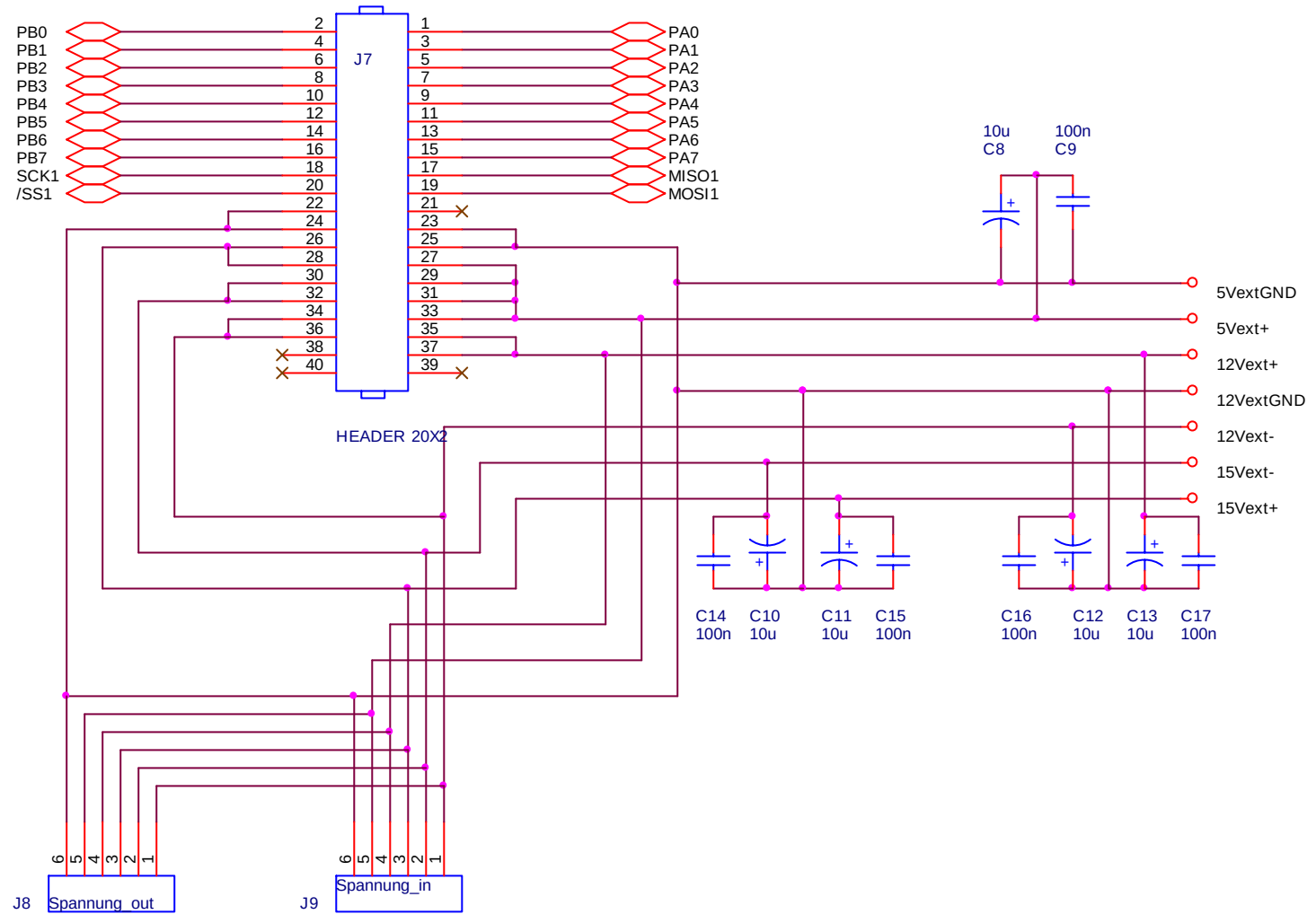
    while(1);
}
//=====
```

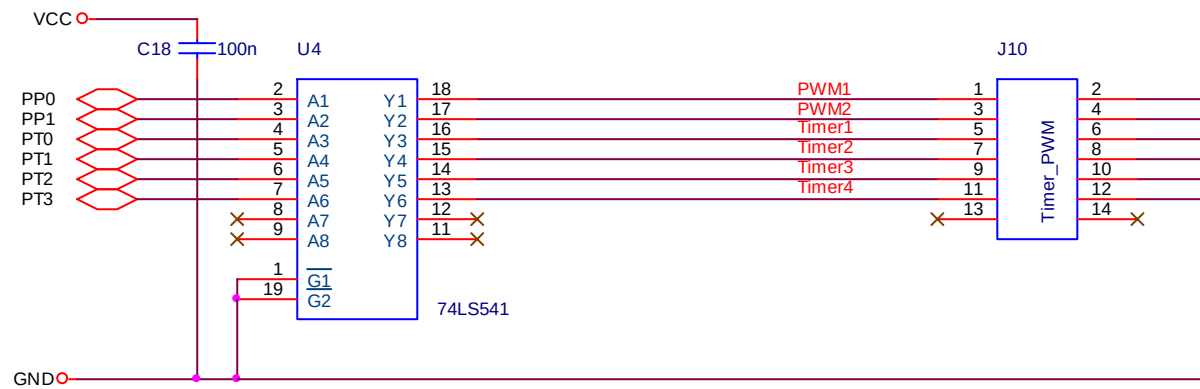
```
    // Timer Setup:
    // Kanaele 0 bis 7 als OC init.
    // Timer Enable Bit setzen
    // Prescaler (=32)
    // Timer Counter Reset Enable
    // Strombegrenzung an fuer 1 bis 4
    // Timer Interrupt Flag loeschen
    // 1 bis 4 als Ausgang
    // 1 bis 4 auf High nach OC an 7
    // 1 bis 4 auf Low nach OC an 1
    // Ausgang 50:50
    // Ausgang 50:50
    // Ausgang 50:50
    // Ausgang 50:50
    // Wenn diese Wert erreicht wird -->
    // Periode neu starten
    // Endlosschleife
```

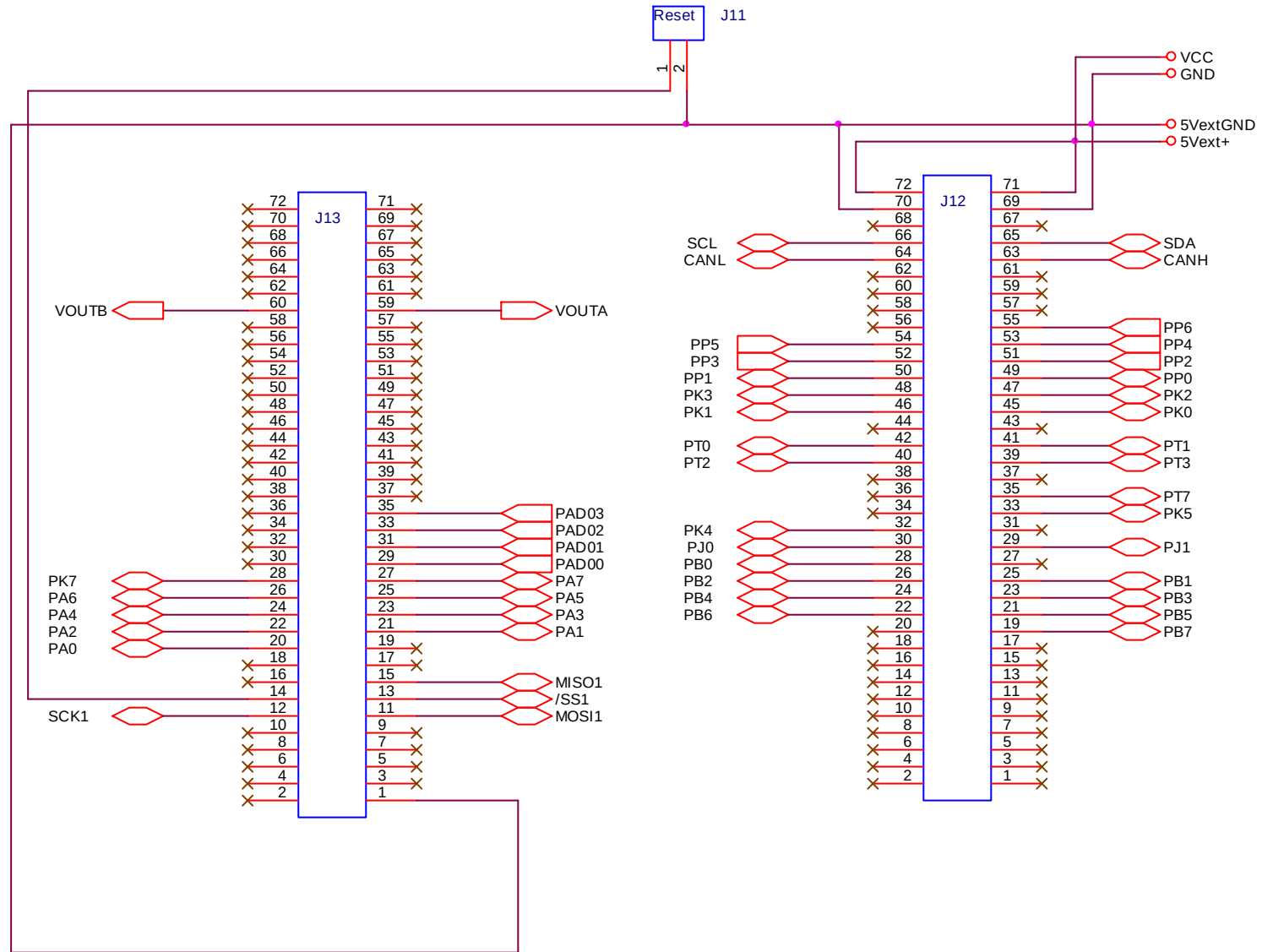


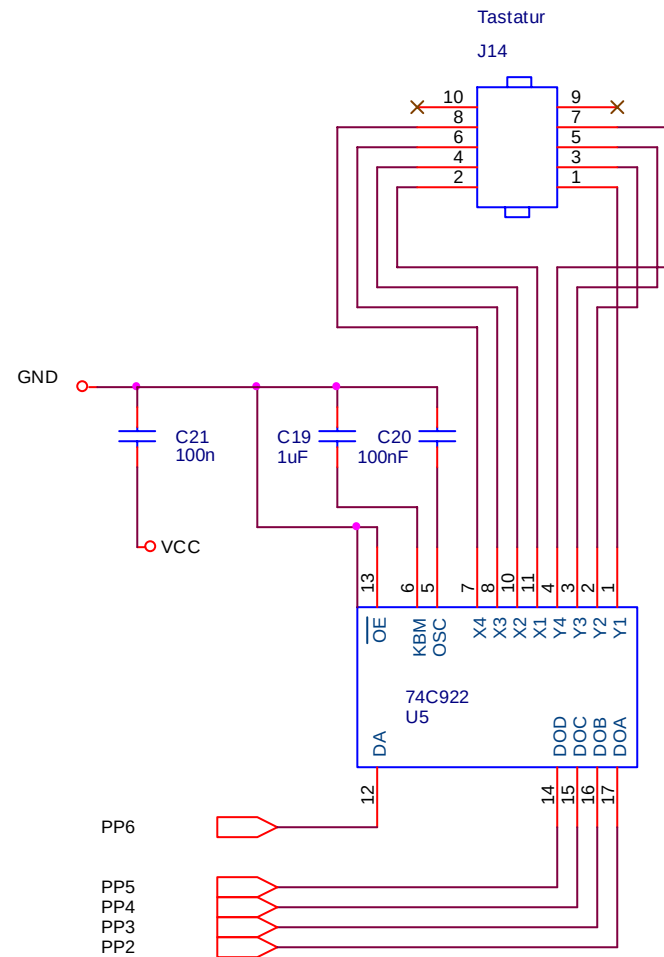


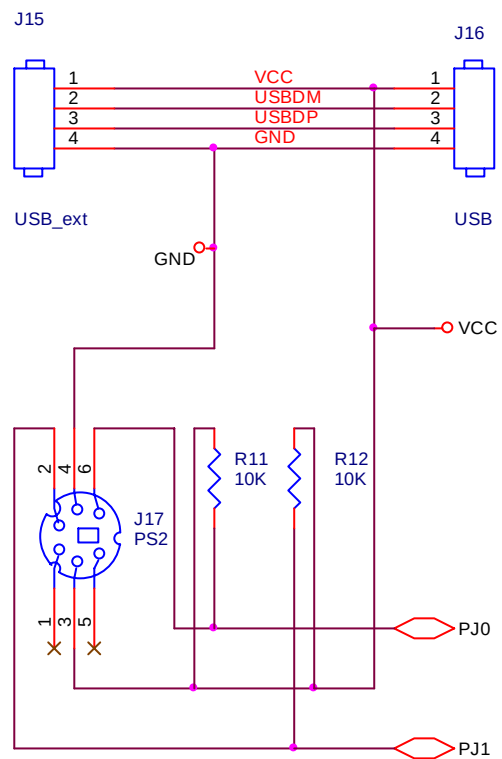


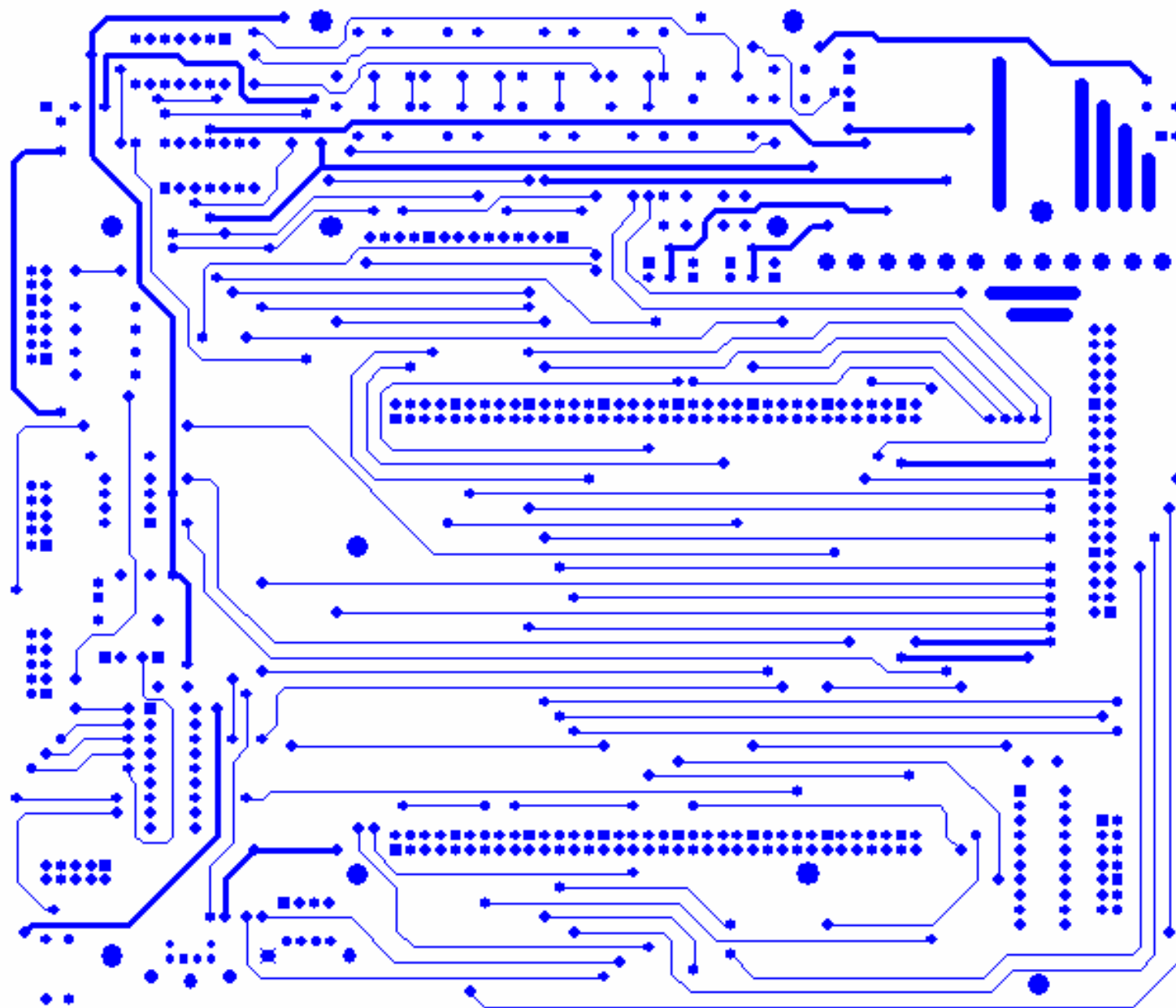


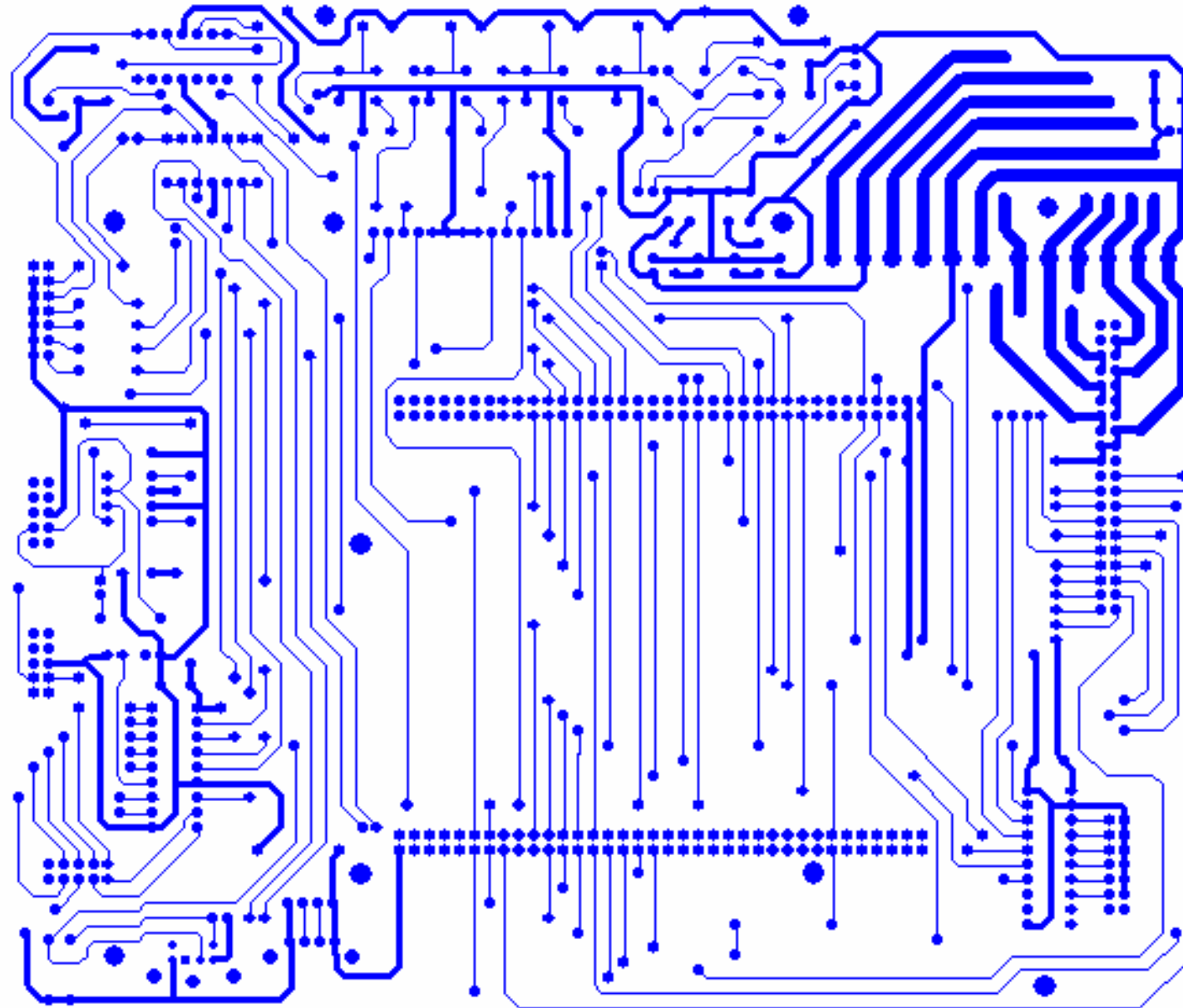


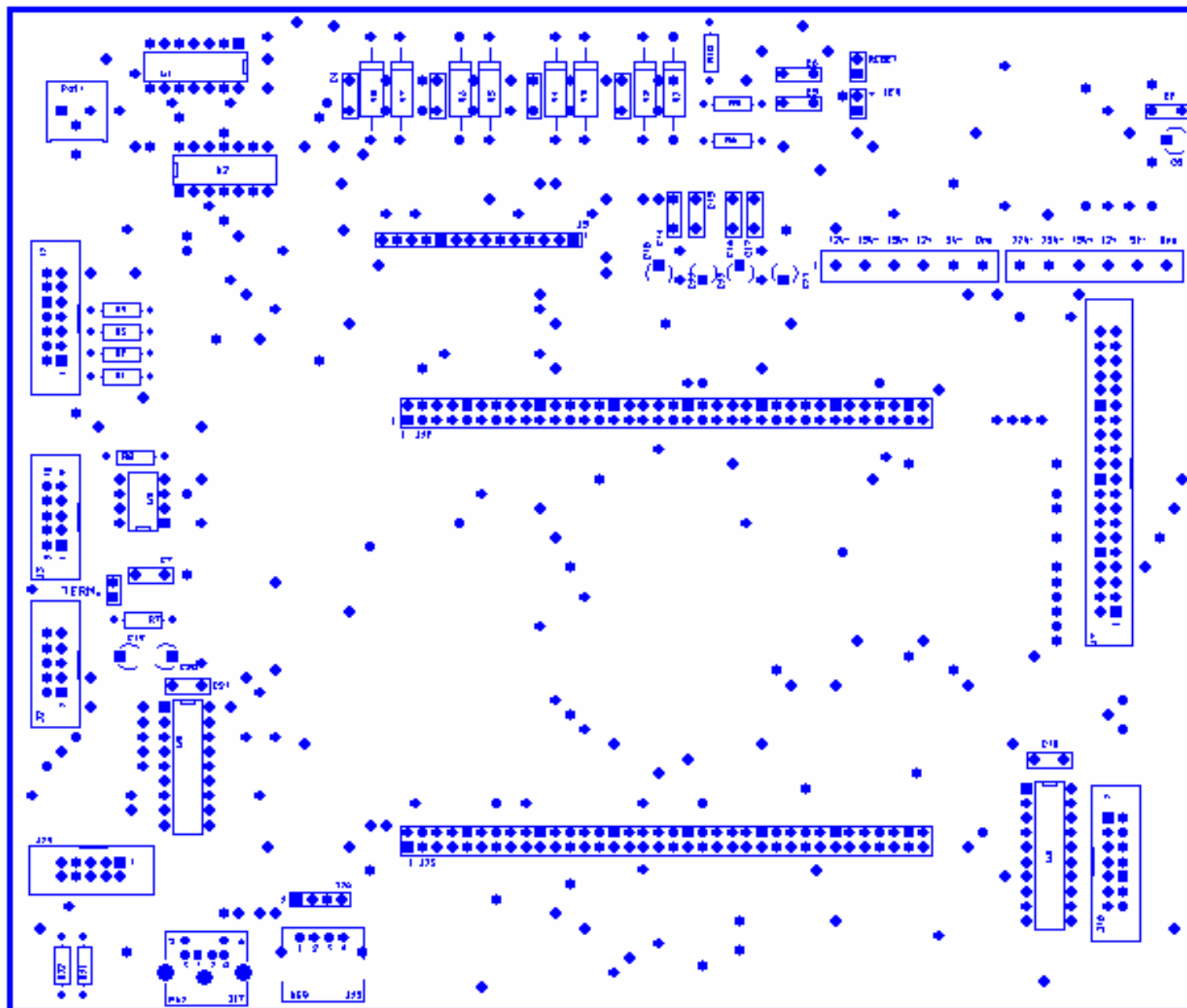












Bauelement	Wert	RM	Bauform	Bemerkungen
C1	10 pF	200	stehend	TP ADW
C2	10 pF	200	stehend	TP ADW
C3	10 pF	200	stehend	TP ADW
C4	10 pF	200	stehend	TP ADW
C5	10 pF	200	stehend	TP ADW
C6	10 pF	200	stehend	TP ADW
C7	100 nF	200	stehend	
C8	10 uF	100	Stehend Elko	
C9	100 nF	200	stehend	
C10	10 uF	100	Stehend Elko	
C11	10 uF	100	Stehend Elko	
C12	10 uF	100	Stehend Elko	
C13	10 uF	100	Stehend Elko	
C14	100 nF	200	stehend	
C15	100 nF	200	stehend	
C16	100 nF	200	stehend	
C17	100 nF	200	stehend	
C18	100 nF	200	stehend	
C19	1 uF	100	Stehend Elko	
C20	100 nF	200	stehend	
C21	100 nF	200	stehend	
D1	Bat 42	700	liegend	
D2	Bat 42	700	liegend	
D3	Bat 42	700	liegend	
D4	Bat 42	700	liegend	
D5	Bat 42	700	liegend	
D6	Bat 42	700	liegend	
D7	Bat 42	700	liegend	
D8	Bat 42	700	liegend	
J1	14 polig		Messerleiste	
J2	10 polig		Messerleiste	
J3	10 polig		Messerleiste	
J4	2 polig		Steckleiste	
J5	14 polig		Steckleiste	
J6	2 polig		Steckleiste	
J7	40 polig		Messerleiste	
J8	6 polig		Klemmleiste	
J9	6 polig		Klemmleiste	
J10	14 polig		Messerleiste	
J11	2 polig		Steckleiste	
J12	72 polig		Steckleiste	
J13	72 polig		Steckleiste	
J14	10 polig		Messerleiste	
J15	4 polig		USB Buchse	
J16	2 polig		Steckleiste	
J17	6 polig		PS Buchse	

Bauelement	Wert	RM	Bauform	Bemerkungen
R1	10 Kohm	400	liegend	TP ADW
R2	10 Kohm	400	liegend	TP ADW
R3	10 Kohm	400	liegend	TP ADW
R4	10 Kohm	400	liegend	TP ADW
R5	10 Kohm	400	liegend	TP ADW
R6	10 Kohm	400	liegend	TP ADW
R7	120 Ohm	400	liegend	
R8	10 Kohm	400	liegend	
R9	20 Kohm		liegend Poti	
R10	20 Kohm	400	liegend	
R11	10 Kohm	400	liegend	
R12	10 Kohm	400	liegend	
U1	LM 348		mit Fassung 14 polig	
U2	LM 348		mit Fassung 14 polig	
U3	VFC424		mit Fassung 8 polig	
U4	74LS541		mit Fassung 20 polig	
U5	74C922		mit Fassung 14 polig	